

Apple Graphics & Arcade Game Design

By
Jeffrey
Stanton



APPLE GRAPHICS & ARCADE GAME DESIGN

BY JEFFREY STANTON

THE BOOK CO.
11223 S. HINDRY AVE.
LOS ANGELES, CA 90045

ACKNOWLEDGEMENTS

A book like this was a long and difficult undertaking. I would like to thank my publisher, James Sadlier for having faith in the book despite its long development time, Don Worth and Lou Rivas for reading the book for technical accuracy, and John Dickey and Gary Kevorkian who edited this book. I would also like to thank Dale Washlake, Phil Wasson, Jim Nitchals, and others who answered many of my graphics questions, and Shannon Hogan who did the cover art from one of my far fetched ideas.

Apple and DOS Tool Kit are trademarks of Apple Computer Co.

Pacman is a trademark of Bally.

Sneakers and Gamma Goblins are trademarks of Sirius Software.

Galaxian is a trademark of Williams.

Scramble is a trademark of Stern.

Space Invaders is a trademark of Namico.

Rip Off is a trademark of Sega.

Threshold and Gamma Goblins are trademarks of On Line Systems.

Missile Command is a trademark of Atari.

Copyright © 1982 by Jeffrey Stanton and The Book Company. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

TABLE OF CONTENTS

INTRODUCTION — 6

CHAPTER 1 APPLESOFT HI-RES — 9

1. Description and Screen Layout
2. Screen Switches and Control
3. Memory Considerations
4. Colors and Background Fill
5. Page Flipping
6. Apple Shape Tables
 - A: Designing Shapes
 - B: Assembling a Directory
7. Graphic Animation Using Shape Tables
8. Character Generators

CHAPTER 2 LO-RES GRAPHICS — 35

1. Introduction
2. Basic Assembly Language
3. Lo-Res Screen Architecture
4. Plotting Dots and Lines
5. Designing the “Breakout Game”

CHAPTER 3 MACHINE LANGUAGE ACCESS TO APPLESOFT HI-RES ROUTINES — 69

1. Description and ROM Addresses
2. HPLOT Shapes and Animation
3. Apple Shape Tables in Animation

CHAPTER 4 HI-RES SCREEN ARCHITECTURE — 87

1. Screen Design and Layout
2. Raster Graphics (Bit Mapped) Shape Tables
 - A: Pros and Cons

- B: Forming Bit Mapped Shape Tables
- C: Shifted Tables for Precise Positioning
- D: Color Problems

CHAPTER 5 BIT MAPPED GRAPHICS — 111

1. Drawing Bit Map Shapes to the Hires Screen
2. Color Problems with Horizontal Movement
3. Screen Erase
4. Selective Drawing Control & Drawing Movement Advantages
5. Interfacing Drawing Routines to Applesoft

CHAPTER 6 ARCADE GRAPHICS — 147

1. Introduction
2. Paddle Routines
3. Dropping Bombs and Shooting Bullets
4. The Invaders Type Game
5. Steerable Space Games
6. Steerable and Free Floating Space Ships
7. Debug Package
8. Laser Fire & Paddle Button Triggers
9. Collisions
10. Explosions
11. Scorekeeping
12. Page Flipping

CHAPTER 7 GAMES THAT SCROLL — 237

1. Games That Scroll
2. Hi-Res Screen Scrolling
 - A: Vertical Scrolling
 - B: Horizontal Scrolling

CHAPTER 8 WHAT MAKES A GOOD GAME — 281

1. What Makes A Good Game
2. Successful Game Examples

INTRODUCTION

A programmer's ability to create Apple graphics can be compared to an artist's ability with a sketchpad or an animator's skill with animation. Each in their own way creates images that are in some way entertaining. The viewer, however, is only interested in the final effect, not the tedious technical process that the artist or programmer had to apply to produce that effect.

The Apple II is a wonderful graphics tool, but unfortunately highly complex to use at any level other than Applesoft BASIC. The scattered magazine articles covering Apple graphics have shown the machine's complexity without presenting an adequate solution to the problem of graphics programming concepts. Those who understand the process and have mastered it are too busy writing programs to share their knowledge.

Magical references like "Raster Graphics" and "Bit Mapping" are spoken of as if they are secret techniques practiced only by the top programmers. Their games, such as "Raster Blaster", "Galaxian", "Sneakers", and "PacMan" have both awed wishful game designers and shown them the limitations of their own programming techniques.

This book will allow you to enter the world of Apple graphics, in which your most imaginative ideas can be animated. The various chapters will attempt to present a comprehensive course in Hi-Res graphics and high speed arcade animation. The major part of this material requires the ability to do assembly language programming. However, since this book was designed to increase the novice programmer's graphics skill, it assumes no prior knowledge of Apple graphics. The book begins with the bare bones graphic techniques of Applesoft BASIC and goes on to teach elementary machine language techniques that will enable the reader to program simple high speed games using the ROM's built in graphics routines.

Bit mapping (or raster graphics) and its use in high speed arcade animation will be covered in great detail. The approach throughout the book is to teach by example. The techniques required to program the three classic game types, (1) Space Invaders, (2) Asteroids, and (3) scrolling games like Defender, are explored. There are sections on paddle control, firing lasers, dropping bombs, explosions and scoring. Page flipping and scrolling techniques are also discussed.

The only requirements for this book are an inquisitive mind, perseverance, and a good assembler. Although prior assembly language programming experience is not necessary, you won't be able to write code without an assembler. The Apple's mini-assembler is totally inadequate for such a task.

I will attempt to explain the ideas in this book through a combination of text, drawings, and flow charts. The concepts in this book may seem easy at times, and somewhat difficult at other times. The Apple with its many idiosyncrasies is a strange beast to master. My advice is to read the book in stages and try the examples. Learn how they work.

While my goal for presenting this material was to educate a new generation of arcade game designers, I dread the proliferation of copy cat games. The world doesn't need an eighth Asteroids game, or a tenth PacMan game. They have been done. I do hope that programmers both young and old will use their imaginations to create something novel and exciting.

A handwritten signature in black ink that reads "Jeffrey Stanton". The signature is fluid and cursive, with the first name "Jeffrey" written in a larger, more prominent script than the last name "Stanton".

JEFFREY STANTON
VENICE, CALIFORNIA
APRIL 16, 1982

PROGRAM LISTINGS AVAILABLE ON DISK

The majority of the code listed in this book is available on diskette to readers who disdain typing long computer programs. The disk is unprotected. The cost of this disk is a nominal \$15.00 plus \$1.50 postage to U.S. residents (foreign orders please add \$5.00 for air mail). California residents add 6% state sales tax (Los Angeles County residents add 6½ % sales tax). Available from The Book Co., 11223 S. Hindry Avenue, Suite 6, Los Angeles, CA 90045. (See order card at back of this book.)

A bit-mapping utility program, which was mentioned briefly in Chapter 4, is available to readers who purchase the above disk for an additional \$10.00 plus tax. It enables the user to design any multi-colored bit-mapped shape on a grid 49 pixels wide by 32 lines deep. The program calculates the subsequent shape table in hexadecimal for both even and odd starting offsets, plus six additional shifted tables if that option is selected. Shapes can be displayed in their actual size and color as well as saved to disk. The program supports a line printer but it is not required.

The Applesoft and machine language object files provided will run on any standard Apple II Computer, but the assembly language source code requires one of three assemblers to interpret them. Big Mac and TED II + assemblers are available from Call A.P.P.L.E. Additionally, Merlin is available from Southwestern Data Systems. These binary source files can also be reformatted for use in other assemblers like Lisa 2.5 or Tool Kit by using a text editor such as Apple Pie.

APPLESOFT HI-RES



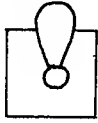
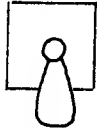
The Apple II computer has the ability to display color graphic images on a video monitor or television screen. It displays these images through a process known as Memory Mapped Output. Various circuits scan specific areas of Random Access Memory (RAM) to determine what should be displayed on the screen. These circuits convert memory information into images containing pixels or dots that are either turned on or off at particular screen positions. Each memory location contains a coded series of instructions for a particular segment of the Hi-Res screen. Thus the hardware maps the image coded in memory to the video screen.

The Apple II computer has two distinct graphics modes. Lo-Res graphics, which occupies the memory space reserved for the text page (\$400 - \$800), has a resolution of 40 dots horizontally by 48 dots vertically. Each dot is very coarse (7 X 8) pixels. Any one of sixteen colors can fill each of the 1920 positions on the screen. Hi-Res graphics, on the other hand, is much more detailed or dense. The resolution is 280 horizontal dots by 192 vertical dots. This gives 53,760 points on the screen. However, only six different distinct colors are available in this graphics mode. (There are actually eight colors including two whites and two blacks.)

Both graphics modes can either be full screen or they can be a mix of graphics and four lines of text at the bottom of the screen. This format reduces the Lo-Res screen to 40 lines and the Hi-Res screen to 160 lines.

Each of the graphics modes has two distinct pages or screens. They reside in specific areas of memory which are hardware set. Each screen can be viewed separately by setting a series of software switches that are located in Read Only Memory (ROM). These are not real physical switches but switches that can be toggled by POKEing values to their ROM reserved memory locations. These switches tell the video hardware to display either text or graphics, Lo-Res or Hi-Res, full screen graphics or mixed text and graphics, and either page 1 or page 2.

When you execute the GR statement in BASIC, the computer turns on the Lo-Res graphics mode, clears display memory so that the screen is black, and defaults to four lines of text at the bottom of the screen. The text window can be eliminated by typing the statement POKE -16302,0, thus giving full screen Lo-Res graphics. Similarly, the HGR statement turns on page one Hi-Res graphics, clears Hi-Res memory so that the screen is black, and defaults to the mixed text and graphics mode. Full screen graphics can be achieved by the statement, POKE -16302,0. And if you wish to view page 2 of Hi-Res

GRAPHICS	FULL SCREEN	PAGE1	LO-RES
-16304 \$C050	-16302 \$C052	-16300 \$C054	-16298 \$C056
			
TEXT	MIXED TEXT & GRAPHICS	PAGE2	HI-RES
-16303 \$C051	-16301 \$C053	-16299 \$C055	-16297 \$C057

memory, the command HGR2 turns it on. The statement POKE -16301,0 sets full screen graphics for page 2.

The principal disadvantage of using HGR or HGR2 is that executing either of these commands clears the Hi-Res page selected, regardless of your wishes. There are times when you have produced a display and want to switch to a full page of text. If you return from text mode through the above commands, your display will be erased.

It is possible to enter the Hi-Res graphics mode without erasing the display screen. If you set the following soft switches which reside in reserved memory locations -16304 through -16297 (\$C050 through \$C057), you can display Hi-Res graphics page 1 without erasing its previous contents.

POKE -16304,0	SETS GRAPHICS MODE
POKE -16297,0	SETS HI-RES MODE
POKE -16300,0	SELECTS HI-RES PAGE 1

Hi-Res page 2 can be displayed with the following commands:

POKE -16304,0	SETS GRAPHICS MODE
POKE -16297,0	SETS HI-RES MODE
POKE -16299,0	SELECTS HI-RES PAGE 2

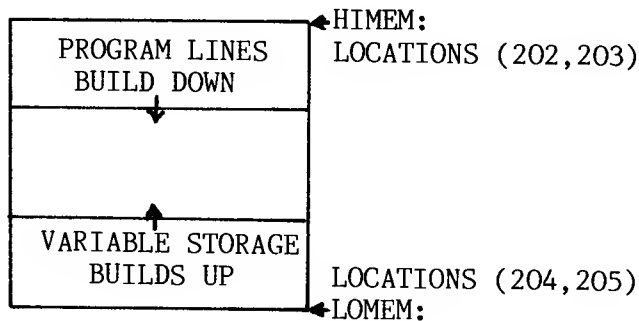
If you wished only to switch displays from Hi-Res page 1 to Hi-Res page 2, only the last command is necessary because the first two commands were previously set.

I should point out that the command "TEXT" will normally return you to page one of the text mode in Applesoft, but may not do so in Integer BASIC. If page two graphics were previously being displayed, the computer would return to page 2 of the text mode. Since this isn't the screen where the commands that you are typing are being displayed, the keyboard would consequently appear to be dead. Page one text can be selected with the statement, POKE -16300,0.

MEMORY CONSIDERATIONS

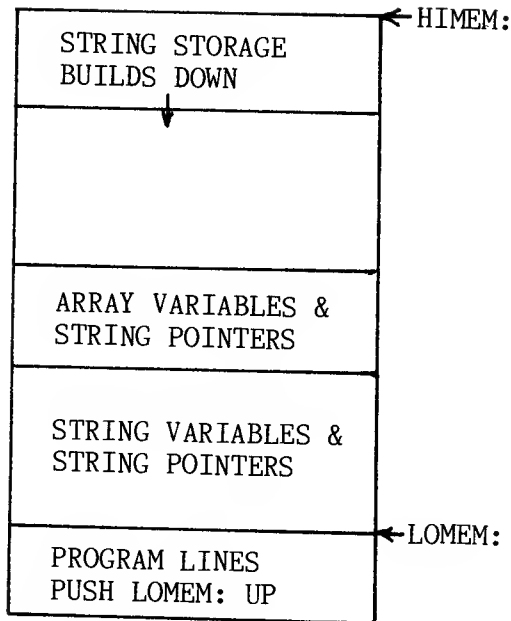
The two Hi-Res screens reside at memory locations 8192 – 16383 (\$2000 – \$3FFF) for page 1, and at 16384 – 24575 (\$4000 – \$5FFF) for page 2. These locations are permanently set. When programming in either BASIC, some considerations must be made as to where you should put your programs so that they don't conflict with the Hi-Res graphics screens.

If we examine an Integer BASIC program memory map below, we see that the program begins at HIMEM:, which is set by the computer to be just below DOS. Variables are stored beginning at LOMEM:, which is normally set just above the text page at location 2048 (\$800). Unless you have some huge storage arrays or a very long program, neither the program nor its variables will cross the Hi-Res screen memory boundary. For safety's sake, it is often better to set LOMEM:16384 (\$4000) so that no conflict could arise. This is especially true if both Hi-Res screens are being used. In that case, set LOMEM:24576 (\$6000).



INTEGER BASIC PROGRAM MEMORY MAP

Applesoft, on the other hand, stores its program just above the text page at 2048 (\$800). Program lines build upwards towards the top of memory. As the program gets longer, LOMEM:, which is the end of the Applesoft program, is pushed upwards. Simple variables and array variables begin just above LOMEM:, and string storage beginning at HIMEM:, builds downward. Thus, setting LOMEM: to a value above the Hi-Res screen would not relocate the Applesoft program nor prevent a long program from occupying the same memory space as the Hi-Res screens.



APPLESOFT BASIC PROGRAM MEMORY MAP

The solution is to set the pointers to the beginning of program text to a value above the Hi-Res screen(s) which you are using. These pointers must be set prior to loading or running the Applesoft program.

The easiest method for accomplishing this is to write an EXEC file which will automatically set these pointers and load or run your program in the proper position. The two pointers that must be set are at locations 103 and 104 decimal, lo byte and hi byte respectively. These are the pointers to the beginning of program text. A reset of the pointers and linkage to either firmware Applesoft ROM or Applesoft in the language card can be assured with a call to the subroutine at 54514 (\$D452). One of the idiosyncrasies of this method requires that a zero byte precede the main program. Therefore the pointers are set one byte higher than requested, and the zero byte is poked into the first position. The following short program will create an EXEC file that will put your Applesoft program in the proper place, free of interference from your graphics.

```

10 D$ = CHR$ (4): PRINT D$;"NOMON C,I,O
20 HOME
25 PRINT "THIS PROGRAM CREATES AN EXEC FILE THAT"
26 PRINT "RELOCATES AN APPLESOFT PROGRAM TO SOME"
27 PRINT "ADDRESS OTHER THA $800 (2048 DECIMAL)"
30 VTAB 6: INPUT "NAME OF APPLESOFT PROGRAM? ";FILE$: IF FI
LE$ = "" THEN 30
40 PRINT : PRINT "ENTER THE DECIMAL ADDRESS FOR THE START":
INPUT "OF THE PROGRAM:";START
45 IF START < 2047 THEN PRINT : PRINT "VALUE MUST BE GREAT
ER THAN 2047": PRINT : GOTO 40
50 PRINT : INPUT "NAME OF EXEC FILE: ";EFILE$
55 S = START + 1:HB = INT (S / 256):LB = S - HB * 256
60 PRINT D$;"OPEN ";EF$: PRINT D$;"DELETE";EF$
65 PRINT D$;"OPEN ";EF$: PRINT D$;"WRITE ";EF$
70 PRINT "FP": PRINT "HOME: POKE 50,128"
80 PRINT "POKE103,";LB;"
85 PRINT "POKE104,";HB;"
87 PRINT "POKE ";START;"",0"
90 PRINT "LOAD ";FILE$
95 PRINT "CALL54514": PRINT "POKE50,255"
100 PRINT "RUN": PRINT D$;"CLOSE"
105 END

```

COLOR & BACKGROUND FILL

There are eight color choices (0-7) on the Hi-Res screen. These are selected by the HCOLOR statement. Since the screen is arranged in alternating columns of either violet-green or blue-orange colors, depending on whether the hi bit is set in a screen memory byte, the absence of color produces two different blacks, and the presence of two adjacent lit pixels produces two different whites. (See chapter 5 for a more detailed explanation.) Thus, only six distinct colors are available. These are listed in the following chart.

COLOR	NUMBER
BLACK	0
GREEN	1
VIOLET	2
WHITE	3
BLACK	4
ORANGE	5
BLUE	6
WHITE	7

Sometimes it is desirable to clear the screen to a background color other than black. This can be accomplished by calling an Applesoft ROM subroutine located at decimal 62454. This clears the screen you used last, regardless of switch settings, to the color most recently HPLOTed. Of course, a call to this subroutine must be preceded by a HPLOT statement. For example, to clear the background to green, try the following:

```
100 HCOLOR = 1:HPLOT 0,0 :CALL 62454
```

PAGE FLIPPING

Using both Hi-Res screens is an effective way of smoothing animation, or creating an image on one screen while viewing the alternate screen. When a group of objects or lines are drawn successively to the screen during an animation frame, the last object drawn is on screen only a fraction of the time that the first object is on the screen. And if there are many large objects, the continuous drawing becomes noticeable.

Page flipping is an effective method to reduce flicker between animation frames. However, one assumes a reasonable animation frame rate of at least 10 frames per second, or the animation appears slow and jerky. The trick to this method is controlling the screen that is drawn to, regardless of the screen switch positions. There is a pointer in zero page, decimal location 230 (\$E6), that sets which screen is plotted to. A POKE 230,32 indicates screen #1, and POKE 230,64 indicates screen #2.

The following example demonstrates the technique. The program HPLOTs thirty random line segments on one screen while the other screen is viewed. It then changes viewing screens to the screen where the image had just been drawn, and erases the opposite screen before randomly drawing thirty new line segments. The result is a series of completed line drawings that change from one image to the next without anyone being aware that they are being drawn elsewhere.

When screen #1 is viewed by toggling the switch with POKE - 16299,0 , the statement, POKE 230,64 , tells the computer to draw to screen #2. Since \$E6 points to screen #2 when the clear screen is called at line 52, it clears screen #2 before plotting our thirty random line segments. When we switch viewing screens to the completed picture with a POKE - 16300,0 ,we reset \$E6 to the opposite screen with a POKE 230,32. Now we are viewing screen #2, and drawing on screen #1.

```
5 X1 = 0:Y1 = 0
10 REM CLEAR BOTH SCREENS
20 HOME : HGR : HGR2 : HCOLOR= 3
30 REM NOW LOOKING AT PAGE #2
40 REM SET DRAWING MODE POINTER (E6) TO SCREEN #1
50 POKE 230,32
51 REM LEAR SCREEN #1
52 CALL 62450
60 FOR I = 1 TO 35
70 X2 = INT ( RND (1) * 280)
80 Y2 = INT ( RND (1) * 192)
90 HPLOT X1,Y1 TO X2,Y2
100 X1 = X2:Y1 = Y2
110 NEXT I
120 REM LOOK AT SCREEN #1 FULL SCREEN
125 POKE - 16300,0: POKE - 16302,0
130 REM SET DRAWING MODE POINTER (E6) TO SCREEN #2
135 POKE 230,64
136 REM CLEAR SCREEN #2
137 CALL 62450
145 FOR I = 1 TO 35
150 X2 = INT ( RND (1) * 280)
160 Y2 = INT ( RND (1) * 192)
170 HPLOT X1,Y1 TO X2,Y2
180 X1 = X2:Y1 = Y2
190 NEXT I
200 REM LOOK AT SCREEN #2
210 POKE - 16299,0
230 GOTO 50
```


As you view the different supposedly random screens, you will notice that the screens appear to repeat every few frames. The repetition, although not perfect, is due to a faulty random number generator in Applesoft. This program graphically illustrates the fault.

A demonstration of the same program without page flipping can be shown. If you take the previous listing and make the following changes, the images can be seen as they are drawn.

```
DELETE LINES 50 & 135
52 HGR2 : POKE-16302,0
125 POKE -16299,0
137 HGR : POKE-16302,0
210 POKE -16300,0
230 GOTO 52
```

APPLE SHAPE TABLES

The Apple II offers a very powerful feature in Applesoft BASIC called shape tables. They are essentially figures or shapes that use tiny vectors to quickly generate their form. They are very flexible in that they can be plotted anywhere on the Hi-Res screen without destroying the background, and they can be scaled (expanded) and rotated. These shapes are often used in animation and game design.

A shape table can consist of up to 255 different shapes. Each shape in the table is generated by outlining it with tiny unit vectors which are all the same length, but may take any of four directions (up,down,left,right). The vectors are placed head to tail until the entire shape is outlined. These vectors can also be of two types: plot vectors or move-without-plotting vectors. Then, using a key, these direction vectors are encoded into a string of hexadecimal bytes which are stored in memory as part of a shape table.

The procedure for creating a shape table isn't difficult, but it is time-consuming and quite prone to error if you aren't careful. The method, due to the nature of its encoding, has several peculiarities that the programmer should be aware of. The most important point, one that is rarely explained, is that the first vector is the position that the shape is drawn when X,Y coordinates are specified. For example, if you wish to draw a square shape to the screen that is two vector units per side, you will prefer to have the shape drawn so that it is centered at the coordinates specified. But if you start your string of vectors at the upper left corner instead of at the center, the shape's center will be at the corner. If the shape is rotated, it will pivot about that point instead of neatly rotating about the square's center. The solution to this misconception is to start at the shape's center and make a move upwards without plotting to the outline of the square's shape.

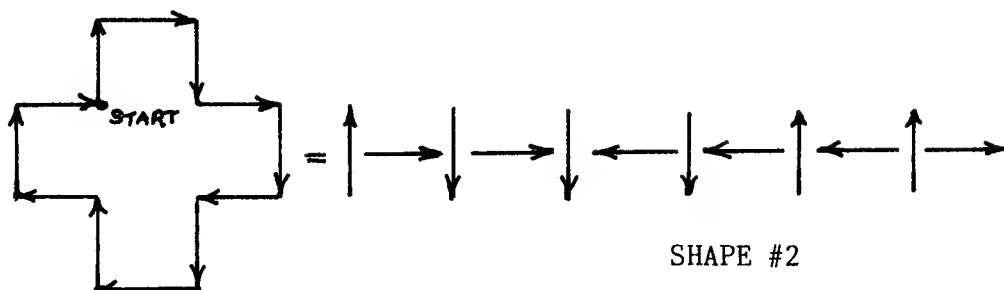
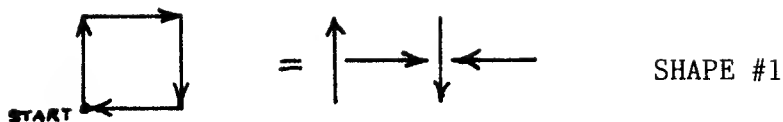
Each shape table byte (8 bits) is divided into three sections. Sections one and two are three bits each and contain any plotting vector. But section three, which contains only two bits, can only hold certain plotting vectors. The three vectors allowed are down, left and right without plotting. Most of the time this section remains unused. This is acceptable, because if section three of the shape definition byte is zero, Applesoft ignores the section and advances to the next byte of the shape.

	SECTION 3		SECTION 2			SECTION 1		
BIT	7	6	5	4	3	2	1	0
M = MOVEMENT BIT	M	M	P	M	M	P	M	M
P = PLOT /NO PLOT BIT								

There is some ambiguity with plotting vectors that are equal to zero. In sections one or two, a zero specifies that you can "move up without plotting", but in section three it means "no movement and no plotting". This also means that you can't have a "move up without plotting" in the third section or it will be misinterpreted.

When all three sections are set to zero, Applesoft interprets it as an end of the shape. This limits the number of "move up without plotting" vectors that can be present in a row. If, for example, sections one and two both contained "move up without plotting" vectors and the next instruction was a plot, section three would be zero also. The value for the byte would be zero, or an end of shape. You can use the "move without plotting" vector in a byte as long as a different plotting vector comes after it. So how do you move upwards several vector units without plotting? By not moving in a straight line. You can move up one, left one, right one, then up one again. This can be repeated a number of times.

All these details may have left your head in a spin, but an example will show that shape tables can be constructed by mere mortals. I should point out that the final table is in hexadecimal, and that once the binary coded plotting vectors for each segment are arranged in groups of two or three within a byte, it becomes easier to divide that byte into two nibbles (4 bits each) for easier encoding.



DRAWINGS OF BOTH SHAPES

SHAPE #1	00	101	100	0010	1100	2C
	00	111	110	= 0011	1110	= 3E
	00	000	000	0000	0000	00
SHAPE #2	00	101	100	0010	1100	2C
	00	101	110	0010	1110	2E
	00	111	110	0011	1110	3E
	00	111	110	0011	1110	3E
	00	111	100	0011	1100	3C
	00	101	100	0010	1100	2C
	00	000	000	0000	0000	00

ASSEMBLING A SHAPE TABLE DIRECTORY

Shape tables are preceded by a shape table directory which contains information concerning the number of shapes in the table, and pointers to the beginning of each shape. The first byte contains the number of shapes (0-255), the second byte is unused, and the remaining pairs of bytes contain the offsets to each shape in the table. The actual number of pairs depends on the number of shapes in the table's first byte.

Although space may be defined for a certain number of shapes when the directory is constructed, there is no rule that says all these shapes need be in the table. Most programmers leave extra space because it is somewhat difficult to expand the table later if extra shapes are needed. A summary of the directory is shown below.

DISPLACEMENT

0	NUMBER OF SHAPES IN TABLE (\$0 -FF)
1	UNUSED
2	OFFSET TO SHAPE 1 LO ORDER BYTE
3	OFFSET TO SHAPE 1 HI ORDER BYTE
	.
2N+2	OFFSET TO SHAPE N LO ORDER BYTE
2N+3	OFFSET TO SHAPE N HI ORDER BYTE
	.
2N+4	PLOTTING VECTORS SHAPE 1
	.
	PLOTTING VECTORS SHAPE N

LENGTH DEPENDS
ON NUMBER OF
SHAPES IN TABLE
(2 BYTES/SHAPE)

If we construct a directory for our previous two shape examples, it takes the following form.

BYTE

0	02	NUMBER OF SHAPES
1	00	UNUSED
2	06	LO BYTE OF OFFSET TO SHAPE #1
3	00	HI BYTE
4	09	LO BYTE OF OFFSET TO SHAPE #2
5	00	HI BYTE
6	2C	} SHAPE #1
7	3E	
8	00	
9	2C	} SHAPE #2
A	2E	
B	3E	
C	3E	
D	3C	
E	2C	
F	00	

This procedure is very time-consuming and, if the shape is complex, prone to error. Fortunately, there are a number of commercial programs that can perform this chore automatically. Most of these, in addition to the standard shape creator, incorporate an editor for merging shapes from several different tables.

Several products that I would recommend are Higher Graphics (Synergistics Software), The Complete Graphics System (CO-Op Software), and Shape Builder and Editor (Telephone Transfer Connection). These packages range in price from \$35 to \$60.

The shape table creator which I've included below lacks an editor for merging, inserting, or deleting shapes. It is also limited to shapes with a maximum size of 25 X 15 pixels. This is inherent in the design, which allows you to define shapes precisely on an oversized grid.

The program is menu-driven and somewhat user-proofed to prevent "bombing" the program in the midst of a hundred-shape-long table, which the user in this case, might have neglected saving periodically to the disk. Once a shape table is initialized, shapes are created one at a time with the command, (C)reate. A starting point is chosen for the shape's center. These values have no relationship to the coordinates where the shape is plotted later, but is the center of the shape and the point about which the shape is rotated with the ROT command. Your shape doesn't have to start there, but can be offset from it or completely surround it.

The current cursor position can be moved by the I,J,K,M keys. If you want to plot a point, press the P key after a move. If you make a mistake, the E key will erase the last plotted point; however, this must be done before the cursor is moved again. Sorry, but it doesn't step back through your keystrokes. When you are finished with the shape, you simply (Q)uit.

When you are returned to the main menu, you have a choice of (V)iewing the shape or (A)dding the shape to the table. Look at the shape first, because if it is incorrect, you can try again with the (C)reate command rather than add it to the table. You can also save the table or load a new table at any time.

This Applesoft program must be relocated above Hi-Res screen page 1. Use the program discussed earlier to create an EXEC file which will reset the pointers. Set the loading address at 16385 decimal. The Shape Creator stores its shape tables at \$800, or 2048 decimal. If you choose to put your tables elsewhere, you must give the program a specific starting location address (e.g., LOAD SHAPE, A\$7000).

Some of the readers who attempt to decipher my code will notice that I stored a value in the second position of the shape table directory. This location is normally unused. I chose to use the location to keep track of the number of shapes currently in the table. The first location contains the maximum number of shapes that the table can hold. This notation is entirely compatible with Applesoft.

```

1 D$ = CHR$ (4):B$ = CHR$ (7)
3 AFLAG = 1:N = 0
5 POKE 232,0: POKE 233,3
14 FOR I = 0 TO 9
16 READ A: POKE 768 + I,A: NEXT I
18 DATA 1,0,4,0,62,36,45,54,4,0
20 TEXT : HOME
24 HTAB 13: PRINT "C O M M A N D S": PRINT
26 HTAB 9: PRINT "(I)NITILIZE SHAPE TABLE": PRINT
27 HTAB 9: PRINT "(C)REATE NEW SHAPE": PRINT
28 HTAB 9: PRINT "(A)DD SHAPE TO TABLE": PRINT
29 HTAB 9: PRINT "(V)IEW SHAPES": PRINT
30 HTAB 9: PRINT "(L)OAD SHAPE TABLE": PRINT
31 HTAB 9: PRINT "(S)AVE SHAPE TABLE": PRINT
32 HTAB 9: PRINT "(Q)UIT": PRINT
33 PRINT "-----": POKE 34,1
7: HOME
34 REM MENU COMMANDS
39 VTAB 19: HTAB 4: PRINT "COMMAND? ";; GET Q$:PK = PEEK (
- 16384): POKE - 16368,0
41 IF PK = 73 THEN 50

```

```

42 IF PK = 67 THEN 100
43 IF PK = 65 THEN 500
44 IF PK = 86 THEN 600
45 IF PK = 76 THEN 65
46 IF PK = 83 THEN 700
47 IF PK = 81 THEN 2000
48 GOTO 39
49 REM INITILIZE TABLE
50 HOME : PRINT : INPUT " NO. OF SHAPES IN TABLE? ";MAX
52 POKE 2048,MAX
54 FOR I = 1 TO 2 * MAX + 1: POKE 2048 + I,0: NEXT I
56 ADDR = 2050 + PEEK (2048) * 2
58 M = 2 + MAX * 2: POKE 2050,M - 256 * INT (M / 256)
59 POKE 2051, INT (M / 256)
60 HOME : GOTO 39
64 REM LOAD SHAPE TABLE
65 HOME : PRINT : INPUT " SHAPE TABLE NAME ? ";NAME$
67 PRINT D$;"BLOAD";NAME$;","A$800"
70 N = PEEK (2049):MAX = PEEK (2048)
76 HOME : IF MAX > N THEN 39
78 PRINT "SHAPE TABLE FULL!": GOTO 2000
99 REM CREATE NEW SHAPE
100 IF N = MAX THEN 450
101 ADDR = 2048 + PEEK (2050 + 2 * N) + 256 * PEEK (2051 +
  2 * N)
102 IF N = 0 THEN ADDR = 2050 + MAX * 2
103 IF AFLAG = 1 THEN N = N + 1
104 POKE 2049,N
106 HGR : HCOLOR= 3: SCALE= 1: ROT= 0:CYCLE = 0
108 FOR X = 0 TO 250 STEP 10: HPlot X,0 TO X,150: NEXT X
110 FOR Y = 0 TO 150 STEP 10: HPlot 0,Y TO 250,Y: NEXT Y
112 HOME : VTAB 22
114 INPUT "ENTER STARTING COORDINATES X,Y? ";X,Y
115 IF X < 1 OR X > 25 THEN 112
116 IF Y < 1 OR Y > 15 THEN 112
117 X = 10 * X - 5:Y = 10 * Y - 5
118 DRAW 1 AT X,Y:XS = X:YS = Y
120 HOME : VTAB 22: PRINT "MOVE PLOT CURSOR WITH KEYS"
122 PRINT "J -LEFT, K -RIGHT , I -UP, M - DOWN"
124 PRINT "P -PLOT ,E -ERASE LAST PLT , Q -QUIT": POKE 36,
41
126 KY$ = "":KSVE$ = "": GOTO 145
128 IF FLAG = 1 THEN 132
130 XDRAW 1 AT X1,Y1
132 X1 = X:Y1 = Y:FLAG = 0

```



```

135 XDRAW 1 AT X,Y
140 KI$ = KSVE$:KSVE$ = KY$
145 GET KY$
150 IF KY$ < > "I" THEN 160
155 SYMBOL = 0:Y = Y - 10: IF Y = > 0 THEN 225
157 Y = Y + 10: CALL - 1052: GOTO 145
160 IF KY$ < > "K" THEN 170
165 SYMBOL = 1:X = X + 10: IF X < = 250 THEN 225
167 X = X - 10: CALL - 1052: GOTO 145
170 IF KY$ < > "M" THEN 180
175 SYMBOL = 2:Y = Y + 10: IF Y < = 150 THEN 225
177 Y = Y - 10: CALL - 1052: GOTO 145
180 IF KY$ < > "J" THEN 190
185 SYMBOL = 3:X = X - 10: IF Y = > 0 THEN 225
187 X = X + 10: CALL - 1052: GOTO 145
190 IF KY$ < > "P" THEN 200
195 FLAG = 1: GOSUB 300: GOTO 135
200 IF KY$ = "Q" THEN 400
205 IF KY$ < > "E" THEN 145
210 HCOLOR= 0:FLAG = 0: GOSUB 300
220 KSVE$ = KI$: HCOLOR= 3: GOTO 130
225 IF KSVE$ = "P" THEN SYMBOL = SYMBOL + 4
230 CYCLE = CYCLE + 1
235 IF CYCLE < > 1 THEN 245
240 BYTE = SYMBOL: GOTO 128
245 IF CYCLE < > 2 THEN 270
250 BYTE = BYTE + 8 * SYMBOL
255 IF BYTE > 7 THEN 128
260 BYTE = BYTE + 8: POKE ADDR,BYTE:ADDR = ADDR + 1
265 BYTE = 24:CYCLE = 2: GOTO 128
270 IF SYMBOL > 3 THEN 280
275 BYTE = BYTE + 64 * SYMBOL
280 POKE ADDR,BYTE:ADDR = ADDR + 1
285 IF SYMBOL = 0 OR SYMBOL > 3 THEN 295
290 CYCLE = 0: GOTO 128
295 CYCLE = 1:BYTE = SYMBOL: GOTO 128
300 FOR Y2 = Y - 3 TO Y + 3 STEP 6: HPLOT X - 1,Y2 TO X + 1
,Y2: NEXT Y2
305 FOR Y2 = Y - 2 TO Y + 2 STEP 4: HPLOT X - 2,Y2 TO X + 2
,Y2: NEXT Y2
310 FOR Y2 = Y - 1 TO Y + 1: HPLOT X - 3,Y2 TO X + 3,Y2: NE
XT Y2
315 IF X = XS AND Y = YS THEN RETURN
320 XDRAW 1 AT X,Y: RETURN
400 IF KSVE$ < > "P" THEN 430

```

```

405 IF CYCLE < > 2 THEN 415
410 POKE ADDR,BYTE:ADDR = ADDR + 1
415 IF CYCLE < > 1 THEN 425
420 BYTE = BYTE + 32: GOTO 430
425 BYTE = 4
430 POKE ADDR,BYTE:ADDR = ADDR + 1
435 POKE ADDR,0:ADDR = ADDR + 1
440 POKE - 16303,0: HOME : VTAB 22: PRINT " (A)DD SHAPE TO
    TABLE IF CORRECT":AFLAG = 0: GOTO 39
450 HOM : VTAB 22: PRINT " SHAPE TABLE FULL!!!": GOTO 39
499 REM ADD SHAPE TO TABLE
500 HOME : IF AFLAG = 1 THEN 540
502 OFF = ADDR - 2048:AFLAG = 1
505 IF N < > MAX THEN 515
510 HOME : VTAB 22: PRINT "TABLE FULL WITH THIS SHAPE!!!"
515 IF N > MAX THEN 550
520 POKE 2050 + 2 * N,OFF - 256 * INT (OFF / 256)
525 POKE 2050 + 2 * N + 1, INT (OFF / 256)
530 GOTO 39
540 VTAB 22: PRINT "NO SHAPE TO ADD!": GOTO 39
550 VTAB 22: PRINT "TABLE FULL CAN'T ADD SHAPE!!!": GOTO 39

599 REM VIEW SHAPES
600 HOME : VTAB 20: INPUT "VIEW LAST SHAPE Y/N? ";Q$
605 IF Q$ = "Y" THEN 627
610 VTAB 20: INPUT "WHICH SHAPE NUMBER TO VIEW? ";K
615 IF K = < N THEN 625
620 PRINT "SHAPE #";K;" DOESN'T EXIST!": GOTO 39
625 M = K: GOTO 630
627 M = N
630 HGR : POKE 233,8: SCALE= 1: DRAW M AT 50,75
635 SCALE= 3: DRAW M AT 165,75
638 VTAB 21: PRINT " SCALE=1 SCALE=3 SHAPE# ";M

640 SCALE= 1: POKE 233,3: VTAB 23: PRINT " PRESS ANY
    KEY!": POKE 36,41
645 GET Q$: POKE - 16368,0: POKE - 16303,0
650 HOME : VTAB 22: IF AFLAG = 0 THEN PRINT " (A)DD SHAPE
    TO TABLE IF CORRECT"
655 GOTO 39
699 REM SAVE
700 HOME : PRINT : INPUT "SHAPE TABLE NAME? ";NAME$
705 PRINT D$;"BSAVE";NAME$;" ,A2048,L";ADDR
710 HOME : GOTO 39
2000 TEXT : END

```

SIMPLE GRAPHIC ANIMATION USING APPLE SHAPE TABLES

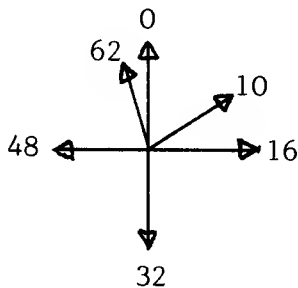
Apple shape tables can be incorporated very easily into games to produce animation. The principle is elementary. A shape is drawn to the screen in one position, then erased before moving it to the next position. If the move is in small increments, and if the animation frame rate is fast enough, the object will appear to have fluid motion. This is exactly how cartoons are animated.

Applesoft has a number of commands which work with shape tables. Any shape in a table can be drawn to the screen with the command, `DRAW N AT X,Y`, where `N` is the shape number in the table, and `X` and `Y` are the screen coordinates to plot the shape. The `DRAW` command plots over the background, thus erasing whatever was there previously. There is an alternate command: `XDRAW`, which exclusive-or's the screen where the shape is plotted. This means if the background is black, the pixels are lit (white) when the shape is `XDRAWn` to the screen, and they revert back to black when `XDRAWn` again. But if the background is white and a white shape is `XDRAWn` to the screen, the pixels are reversed, so that the shape becomes black. Similar complementary effects occur if the background color is green, blue, orange or violet.

Shapes can be rotated with the `ROT` command or scaled with the `SCALE` command. Values can range from 0-255. Values for both `SCALE` and `ROT` must be set to some value before drawing a shape for the first time.

When a shape is drawn at a scale larger than one (`SCALE = 0` is equivalent to 256), the computer will draw more than one point for each unit vector. If the scale is four, four points will be drawn for each single plotting vector.

Although rotation angles can range from 0-63, the actual number of rotation angles depends on the shape's scale. When the scale is set to 1, rotations can only occur in 90 degree increments (0 = 0 degrees, 16 = 90 degrees, 32 = 180 degrees, and 48 = 270 degrees). Shape rotations at `SCALE = 2` can be incremented by 45 degrees, and by specifying `SCALE 5` or greater, all 64 rotational angles are possible.



ROTATION ANGLES

When a shape is plotted to the screen, Applesoft needs to know the location of the stored shape table. Locations 232 and 233 decimal contain the starting address of the table, lo byte first. Thus, if the table were stored in memory at \$300 or 768 decimal, Applesoft would be informed with `POKE 232,0 : POKE 233,3` (00 being the lo order byte and 03 being the hi order byte).

It is important to find a safe spot in memory for your table, a place where it won't be overwritten by either the Applesoft program or its variable storage space. Short shape tables can be placed in page three of memory (locations \$300 - \$3CF) as long as you aren't using those locations for any other machine language routine, such as sound. An alternate location would be above the string storage space at HIMEM:. This involves resetting the pointers to a lower value. Addresses 115 and 116 (\$73 and \$74) contain the latest HIMEM: values, stored as lo byte first. The new address can be computed by the following statements.

```
PRINT PEEK(116)*256 + PEEK(115) - X
```

where X is the length of the shape table.

```
HI = INT ( HIMEM/256 )
```

```
LO = HIMEM - 256*HI
```

Then use the statements `POKE 116,HI : POKE 115,LO` to reset HIMEM:.

The shape table is then BLOADED at this address and locations 232 and 233 are set to point to the table.

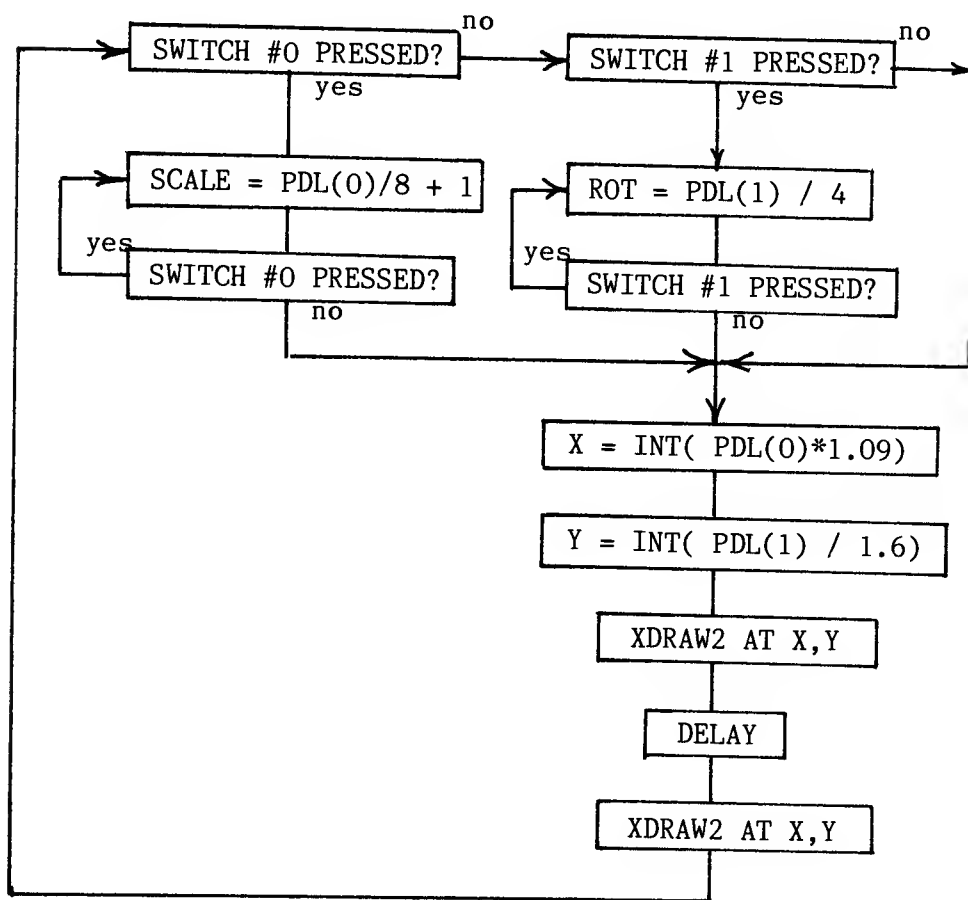
Sometimes it is best to illustrate a concept with an example. Many animated shapes like gun crosshairs are moved around the screen by paddle or joystick control. We can take shape #2, which is shaped like a cross, from our previous shape table example, and XDRAW it to the screen at a position determined by the settings of the two paddles. Remember that if you XDRAW a shape to the screen the first time, the shape appears. But if you XDRAW a shape that is on the screen, it will disappear.

The paddles in this example do more than just position the crosshair. If button #0 is depressed, the paddle setting changes the SCALE, and if paddle #1 is depressed, that paddle setting varies the ROT (rotation). Thus, you are able to observe the various effects that occur when varying the drawing parameters. Wrap-a-round is the most observable effect. This occurs when part of a shape crosses the screen's borders. This feature, which is performed automatically, can be either a help or a hindrance depending on the desired effect. There are times when you would like your shape to exit cleanly off one side of the screen without appearing at the opposite side. In those cases, you will have to test the screen coordinates so that wrap-a-round doesn't occur. Others who have, for example, a freely-floating spaceship, will be pleased by the convenience.

For convenience sake, I poked the shape table into memory at location 768

(\$300) with a FOR-NEXT loop that reads the values in a DATA statement. The hexadecimal shape table values have been converted to decimal values for the data. The alternate method is to enter the monitor and put the values into memory directly at \$300, then BSAVE the table (BSAVE SHAPE, A\$300,L\$10 or BSAVE SHAPE, A768,L16).

Several of the paddle-controlled variables are scaled in the program. Paddle values range from 0 - 255. To obtain X coordinate values, which range from 0-279, the paddle values are multiplied by 1.09, and Y values are divided by 1.6 to keep them within the screen boundaries of 0-191. The SCALE was also trimmed to values 0 to 32 by dividing by 8. I think you will find the code and the accompanying flow chart clear.



```

1 POKE 232,0: POKE 233,3
5 FOR I = 0 TO 15: READ V: POKE 768 + I,V: NEXT I
10 HGR : POKE - 16302,0: HCOLOR= 3
15 SCALE= 4: ROT= 0
20 BUT = PEEK ( - 16287): IF BUT < 128 THEN 60
30 SALE= INT ( PDL (0) / 8 + 1)
32 XDRAW 2 AT X,Y
34 FOR DE = 1 TO 50: NEXT DE
36 XDRAW 2 AT X,Y
40 BUT = PEEK ( - 16287): IF BUT > 127 THEN 30
50 GOTO 90
60 BUT = PEEK ( - 16286): IF BUT < 128 THEN 90
70 ROT= INT ( PDL (1) / 4)
72 XDRAW 2 AT X,Y
74 FOR DE = 1 TO 50: NEXT DE
76 XDRAW 2 AT X,Y
80 BUT = PEEK ( - 16286): IF BUT > 127 THEN 70
90 X = INT ( PDL (0) * 1.09)
100 Y = INT ( PDL (1) / 1.60)
110 XDRAW 2 AT X,Y
120 FOR DE = 1 TO 50: NEXT DE
130 XDRAW 2 AT X,Y
140 GOTO 20
200 DATA 2,0,6,0,9,0,44,62,0,44,46,62,62,60,44,0

```

Drawing shapes to the screen with XDRAW commands isn't the only method of drawing if erasing background is not a concern. The DRAW command works just as well for putting an object on the screen. The XDRAW command is still used for erasing the object. However, the DRAW command doesn't work properly at certain combined rotation angles and scale factors. This can be demonstrated in the last program by changing the XDRAWS in lines 32, 72 and 110 to DRAW commands. Now if the program is run, pixels from the shape sometimes aren't erased at some rotation angles with large scale factors. Thus, it is safer to always use the XDRAW command.

CHARACTER GENERATORS

Character generators are designed to assist the programmer in placing text on the Hi-Res screen. Their ability to mirror the print functions on the text screen makes them extremely easy to use from BASIC programs. Once the character generator is engaged (usually by a CALL to its starting address) any print statements within the BASIC program are printed on the Hi-Res screen instead of the text page. The HTAB and VTAB functions are fully supported, so that Hi-Res text can be accurately positioned.

Since the character set is in memory rather than in a ROM chip on the keyboard, character sets can be changed at will. An Old English or Gothic character set could easily be substituted for the standard ASCII character set used in the ROM.

This versatility in character set design has led to users creating character sets consisting of playing cards, alien monsters for games, or electrical symbols used in schematics. While each character is only 7 X 8 pixels, groups of characters can be arranged in a block to form larger shapes. A playing card could easily consist of nine different characters, forming a three by three block. If the Q W E A S D Z X C letters were used to define the queen of hearts, printing them to the screen in the following form would produce the playing card:

```
QWE
ASD
ZXC
```

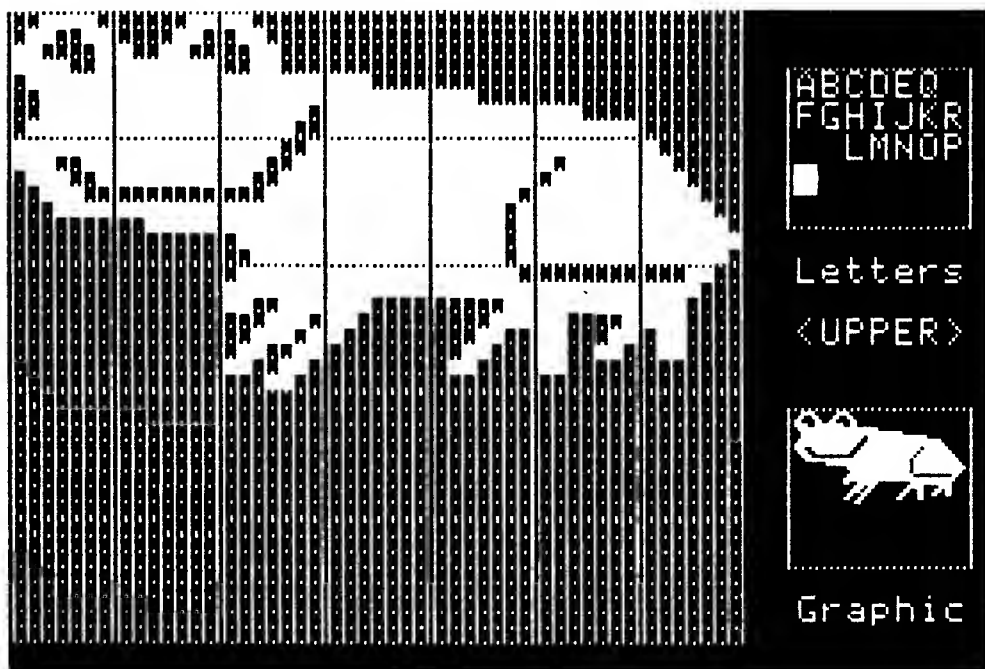
With 96 different characters available in one character set, you could easily represent the 13 card values, if two of the diagonal character elements defined the suit.

Many programmers have taken advantage of the high speed drawing ability of these machine language character generators to do animated graphics. Since sequences of characters representing shapes can be rapidly "printed" on the Hi-Res screen, each animated frame consists of characters "printed" at a new position.

Animating with character generators is relatively easy; however, it does have several disadvantages. First, the speed advantage gained by the machine language routine is badly offset by interfacing it with Applesoft. BASIC programs need to be compiled into machine code in order to produce marginal frame rates. Second, animation appears to be jerky due to the nature of the character position boundaries. There are only 40 horizontal positions and 24 vertical positions for placing a character on the Hi-Res screen. Since characters can't be drawn in-between positions, they tend to jump 8 pixel positions vertically and 7 pixel positions horizontally. Lastly, as a rule, character generator animation lacks color. Most limit color because of the peculiarities of the Hi-Res screen. If, for example, a green character were "printed" in column one, it would appear violet in column two. This would require two character sets to

compensate for this annoying effect between even and odd columns. It is easier to buffer the color to white.

The need to design new character sets has spawned a number of commercial character set editors and character set generators. One versatile package is included in the DOS TOOL KIT that is available from Apple Computer Incorporated. It has a program called "Animatrix" that enables you to construct shapes consisting of a number of user-defined characters. The illustration below shows a shape drawn on the enlarged grid, while the display in the upper right shows which characters these represent. When the character set is attached to their character generator (also in this package), animated drawings or games can be produced. They include an example of an animated game in which a joystick-controlled frog leaps in the air to catch passing butterflies.



ANIMATRIX DRAWING

Other available character generators are HIGHER TEXT from Synergistics Software and SCREEN MACHINE from Softape. Neither is suited for large character animation, but HIGHER TEXT can produce very nice color text displays.

HOW CHARACTER GENERATORS WORK

Character generators incorporate high speed machine language routines that calculate the character's position, then draws it on the screen one byte at a time. Characters consist of eight bytes in memory, where each byte represents the on/off positions of seven adjacent pixels. Each character is 7 pixels wide by 8 pixels deep. There are 96 characters in a set, each eight bytes in length, for a total of 768 bytes of memory.

The program has an index to the character set. Each character fits in a particular position within the set depending on its ASCII assigned value. The character numeric values range from decimal 160 to 255, including both upper and lower case characters. When the character generator begins processing the PRINT statement within the BASIC program, it reads a character, determines its ASCII value, then indexes to the proper eight bytes in its table to obtain the character shape bytes to be drawn to the screen. For example, the program says to print an H, which is interpreted as the ASCII character 200. That character is 40 characters past the tables first character value. Therefore, the H shape begins 40 X 8 bytes into the character set storage table. Now those eight bytes which will be plotted on the screen don't have to represent an H. They may have been redefined with a character editor to be a section of a much larger shape.

\$800	00	00	00	00	00	00	00	00	ASCII 160 (blank)
	
	
	
\$900	1C	22	2A	3A	1A	02	3C	00	ASCII 192 (@)
\$908	08	8C	14	92	3E	22	22	00	ASCII 193 (A)
\$910	1E	22	22	1E	22	22	1E	00	ASCII 194 (B)
	
	

$$\text{Char A} = 2048 + (193-160)*8 = 2312 (\$908)$$

Most character generators use control characters to set various modes. The Apple II lacks a true lower/upper case shift key; control characters are used for this function. Sometimes, control characters are used to put the user in “Block Mode”. This saves inserting numerous VTABs and HTABs when printing a multi-character shape such as playing cards. Other control characters are often used to clear to the end of a line or even an entire page. This facilitates erasing the old characters before drawing new ones on the screen.

Screen animation is obtained by drawing the characters at one position, then moving them to the next position. Unlike Apple shape tables, you don’t need to XDRAW to erase characters. Instead, leading or trailing blanks are added to help erase characters from the old string that may not be erased when drawing the new string. It is equivalent to using a DRAW command, with spaces inserted on either side of the shape. The other alternative is to erase the character shape entirely using blanks. This method is more likely to increase screen flicker since an extra step is involved.

The TOOL KIT character generator has one feature not found in other packages. It has the ability to preserve background while drawing characters. A good example of this is the demo game, RIB * BIT. The character generator stores the background picture on Hi-Res page two, and ORs the characters against it while drawing on Hi-Res page one. This technique also facilitates erasing the characters in their previous position. One is relieved of the task of printing blanks to the Hi-Res screen before repositioning the character shape.

In summation, although a character generator is capable of animating simple games from BASIC for beginners, it doesn’t offer the speed, flexibility, color, and smoothness that is required for quality arcade games. Although character generators have their place, there are better methods presented later in this book.

LO-RES GRAPHICS

The words, machine language and/or assembly language, evoke visions of indecipherable code to the novice BASIC language programmer. The code looks unfamiliar. But so was BASIC when it was first learned. While BASIC has its roots in the English Language and algebraic expressions, assembly language appears to consist of unfamiliar op codes or mnemonics that are used in conjunction with an unfamiliar base 16 number system called hexadecimal.

It is my intent in this chapter to teach you the fundamentals of assembly language programming by comparing it to similar code written in BASIC. Rather than try to teach all aspects of the language, I'll concentrate only on the operations needed to do simple Lo-Res plotting and, later, additional operations to enable you to write a Lo-Res Breakout game.

A good assembler is needed to write assembly language programs. Although owners of Apple II Integer BASIC machines have mini-assemblers built-in, they don't offer the flexibility needed to write anything other than short programs. A good assembler allows you to enter assembly language code by line number and later edit, insert or delete particular lines. Since any line of code can have a label in its first field, the assembler will automatically calculate the branches or "GOTOs" to lines referenced with these labels. Also, if you wish to store a value in a variable called "ZAP", the assembler which assigns a memory storage location for the variable, and will automatically furnish the correct memory address for any subsequent store or load operations using that variable.

Readers who already own assemblers may use the one they have. For those of you who are new programmers, I would recommend one of two types of assemblers. One type of assembler evolved out of the Apple Computer organization and the Apple Puget Sound Programming Library (CALL - A.P.P.L.E.). These are mostly co-resident assemblers, wherein both the assembler and text editor reside in memory simultaneously. They are marketed under names like TED II + , BIG MAC , MERLIN, and TOOL KIT. Only the TOOL KIT is the exception. It is disk-based and loads either the assembler or text editor to memory. Its prime advantage lies in writing larger programs; however, its disadvantage is that it is time-consuming to shift files back and forth to the disk when testing short programs. I chose and used BIG MAC for writing the programs for this book. The other popular assembler that I would recommend is the LISA series by Randall Hyde. It is a co-resident assembler with a mediocre text editor and fast assembler, but its mnemonics are not completely compatible with the other assemblers. It also complements Randy's "Using 6502 Assembly Language" book, which I would recommend

reading for a more comprehensive introduction to assembly language programming. However, it does not cover graphics.

BASIC ASSEMBLY LANGUAGE

The Apple II contains a central processing unit (CPU), a 6502 microprocessor. It accepts instructions to perform various operations, like taking a value and storing it somewhere in memory, adding a number to another number located in one of its internal registers, or comparing two values. What makes programming in assembly language rather difficult (or at least tedious) is that it can only execute one tiny instruction at a time, and only perform its operations in three internal registers. These three addressable registers are known as the X register, Y register and Accumulator. Each can hold eight binary digits called bits, which are individually valued at 0 or 1. The eight bits, collectively called a byte, have values ranging from 0 to 255 decimal or (\$00 to \$FF in hexadecimal notation).

Essentially, the computer, which is an eight bit microprocessor, can manipulate data whose values range from all eight bits off (00000000) to all eight bits on (11111111). The average person has great difficulty in thinking of values represented by 0's and 1's. Fortunately, someone invented a number system called hexadecimal, which is base 16 instead of binary or base 2.

Since 16 is $2 \times 2 \times 2 \times 2$, we can divide our eight bits into two four bit groups. If you determine each of the decimal equivalents of all the combinations of base two representations, you obtain the following table. These values range from 0 to 15 decimal. In the hexadecimal numbering system, values above 9 are represented by the letters A - F. In order to prevent confusion between decimal and hexadecimal numbers, hexadecimal numbers are preceded by a "\$".

BINARY	DECIMAL	HEXADECIMAL
0000	0	\$0
0001	1	\$1
0010	2	\$2
0011	3	\$3
0100	4	\$4
0101	5	\$5
0110	6	\$6
0111	7	\$7
1000	8	\$8
1001	9	\$9
1010	10	\$A
1011	11	\$B
1100	12	\$C
1101	13	\$D
1110	14	\$E
1111	15	\$F

Hexadecimal numbers are very much like decimal numbers. They can be added and subtracted in like manner. The only difference is that instead of having units, tens and hundreds, etc, the hexadecimal numbers have units, sixteens and 256's, and so forth. Each successive digit is 16 times the position to the right instead of ten times as in our decimal system.

DECIMAL	HEXADECIMAL
1 6 5	\$ 1 3 A
1 HUNDRED	1- 256
6 TENS	3 SIXTEENS
5 ONES	A - ONES
1 x (100) = 100	1 x (256) = 256
+ 6 x (10) = 60	+ 3 x (16) = 48
+ 5 x (1) = 5	+ A x (1) = 10
-----	-----
165 DECIMAL	\$ 13A = 314 DECIMAL

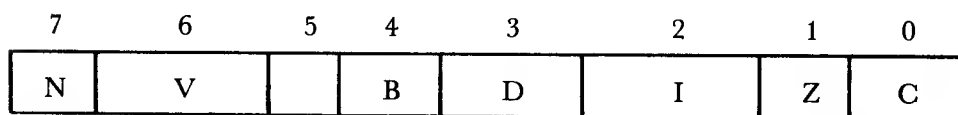
Hexadecimal numbers are used to address the Apple II's 48000 + memory locations. Each group of 256 bytes (\$00 - \$FF) is called a page, starting with page zero. In 48K Apples, memory is directly addressable from locations \$0000 to \$BFFF (0 - 49050). Locations above \$BFFF are also addressable, but these locations don't contain RAM. These locations, from \$C000 - \$FFFF, either address physical connections like the speaker and game switches at locations \$C000 - \$CFFF, or address the ROM (Read Only Memory) beginning at \$D000 and extending to \$FFFF. The latter area contains machine language monitor routines and either Integer or Applesoft BASIC, depending on whether you have an Apple II or Apple II Plus.

MEMORY MAP

192	\$C000 - \$FFFF	HARDWARE & ROM
191		
150	\$9600 - \$BFFF	DOS
149		
96	\$6000 - \$95FF	FREE RAM
95		
64	\$4000 - \$5FFF	HI-RES PAGE #2 OR FREE RAM
63		
32	\$2000 - \$3FFF	HI-RES PAGE #1 OR FREE RAM
31		
12	\$C00 - \$1FFF	FREE RAM
11		
8	\$800 - \$BFF	FREE MEMORY OR PAGE #2 TEXT & LO RES
7		
4	\$400 - \$7FF	PAGE #1 TEXT & LO RES
3	\$300 - \$3FF	MONITOR VECTOR LOCATIONS
2	\$200 - \$2FF	GETLN INPUT BUFFER
1	\$100 - \$1FF	SYSTEM STACK
0	\$00 - \$FF	ZERO PAGE - SYSTEM VARIABLES
PAGE	HEX RANGE	USEAGE

The lowest eight pages of memory, locations \$0000 to \$07FF, are very important; programs should not be stored there. The upper four pages of this section of memory, \$0400 to \$07FF, are the memory locations of the text screen page. Storing values in these locations directly affects the text display. Page two, \$200 to \$2FF, is the keyboard buffer. Inputting data from the keyboard tends to wipe out stored data here. Page one, \$100 to \$1FF, is called the stack. It is used by a special purpose register in the 6502 microprocessor for keeping track of return addresses when calling subroutines. This scratch area for the Stack Pointer is sometimes used for temporary register storage. Page zero, \$00 to \$FF, is a very special area. There are a number of zero page addressing instructions. These instructions are two bytes long instead of the usual three, because they address a memory location from \$00 to \$FF instead of \$0000 to \$BFFF. The latter takes an extra byte to address the larger addresses. Also, these instructions execute faster. Page zero is used extensively for variable storage by the monitor, BASIC interpreters, and DOS. Only some of these memory locations are free for your use. You should consult the chart in the Apple Reference manual for usable locations.

When a microprocessor processes a machine language program, it keeps track of which instruction it is executing with an internal 16 bit register called the program counter. The program counter contains the current address of the instruction that is being processed. When the computer finishes with an instruction, it sets a flag or condition in a seven bit, Program Status Word, which is a register. For example, if you want to test if a value in the Accumulator is equal to zero, you can compare the Accumulator to zero. If true, the zero flag will be set and the instruction Branch Equal to Zero (BEQ) will be executed. Other flags that can be set are the carry flag, overflow flag, and the negative flag. A diagram of the Program Status Word is shown below.



SIGN OVERFLOW

BREAK DECIMAL INTERRUPT ZERO CARRY

PROGRAM STATUS WORD

The 6502 microprocessor accepts only machine language instructions. These are called op-codes. When the computer encounters a \$4C, it performs a equivalent to a GOTO in BASIC. The machine language instruction \$4C 00 08 tells the computer to jump to memory location \$800. (Remember, addresses require two bytes with the low order byte containing \$00 and the high order byte, \$08 — in effect, the reverse order of the actual values. Unfortunately,

machine language is difficult to remember, so programmers invented a substitute called Assembly language, wherein each op-code is assigned a mnemonic such as JMP, BRK, and LDA. The above example looks like this: JMP \$0800.

If you were to type the following machine code into the monitor, you would see how the monitor disassembler interprets the code, as in the following example:

```
>CALL-151
*800:A9 05 8D 00 09 CE 00 09 AD 00
      09 C9 00 D0 F6 60 < CR >
```

If you enter a 800L from the monitor you will see the following:

```
0800 A9 05      LDA #$05
0802 8D 00 09   STA $0900
0805 CE 00 09   DEC $0900
0808 AD 00 09   LDA $0900
080B C9 00      CMP #$00
080D D0 F6      BNE $0805
080F 60         RTS
```

The disassembler translates the machine code to easier understood mnemonics. In the first line of code, LDA is the mnemonic for Load Accumulator. It is the instruction for the 6502 to load the Accumulator with an immediate value -in this case, \$05. The # sign signifies that it is an “immediate” instruction; the (\$05) is the data portion of the instruction. The STA in line two is an “absolute” instruction. It specifies the address in memory for storing the byte of data that is in the Accumulator.

The difference between “immediate” and “absolute” instructions is an important point. Let us take the example LDA #\$05. In this “immediate” instruction, the computer takes the operand (\$05) as a value and places it in the Accumulator. However, with LDA \$05, which is an “absolute” instruction, the computer takes the operand as an address from which to load data in the computer. In both cases, we get a value in the Accumulator. You can tell the modes apart because “immediate” instructions have a # sign before the operand.

You might wonder, what does this code do? It puts the value of 5 in memory location \$900. Line two stores it there, then the value of that memory location is decremented by one in line three. It is then reloaded into the Accumulator to be compared against the value zero. If it is zero it falls through to a return-from-subroutine and ends; but if it isn't zero it branches back to memory location \$805. That location tells the computer to decrement the value in \$900 once

again. The code will perform this small loop until the value in \$900 becomes zero. At that time, the test for a zero becomes true and the program returns to whatever called it. In our case, we called the code from the monitor - thus it returns to the monitor. If we had called it from within a program, it would have returned to the appropriate place in the code to continue the program.

Does it work? First, type 900:AA <CR> to place something in that memory location, then type 800G <CR> from the monitor. The code will return you back to the monitor when it finishes. Type 900 <CR> and a 00 is returned. This is the value in memory location \$900. If you have an Integer machine that has STEP and TRACE, you can do a 800S <CR> instead, followed by a S <CR> each time and watch the code single step. The value in the Accumulator is the first value displayed. When it finally reaches zero the program will reach the RTS and finish.

This program has a direct analogy to the following BASIC program:

```
10 X = 5
20 X = X - 1
30 IF X <> 0 THEN 20
40 RETURN
```

The major differences between the two programs is that in assembly language there are no line numbers, and you have to take care of every detail. BASIC automatically assigns the storage locations of all variables and the location of each instruction in memory. In assembly language programming, we have to assign the X variable to memory location \$900 and have to calculate the relative branch or GOTO so that it references the memory location \$805. This is done by branching back \$F6 bytes, or -8 bytes, to the proper address. Yet, many of these details can be greatly simplified if we use an assembler to do our programming.

The same program using an assembler looks like the following:

	LINE #	LABEL FIELD	INSTRUCTION FIELD	COMMENT FIELD
	1		ORG \$800	;ASSEMBLE CODE AT \$800
	2		OBJ \$6000	
	3	X	EQU \$900	;X IS STORED AT \$900
0800:	4		LDA #\$05	
0802:	5		STA X	
0805:	6	LOOP	DEC X	;X = X - 1
0808:	7		LDA X	
080B:	8		CMP #\$00	
080D:	9		BNE LOOP	
080E:	10		RTS	

The assembler generates identical machine code, but many of the tedious details are simplified. Once X is equated to the memory location in line 3, references to that variable in lines 5 through 7 are handled automatically. If X were assigned to a different memory location because our program was lengthened, you would only have to change line 3. Also, labels are allowed. They act like line numbers in BASIC. Since the assembler assigns the line of code labeled LOOP to a particular memory location, it can calculate the correct relative branch automatically when it encounters line 9 during assembly. The ORG and OBJ in lines one and two are pseudo-opcodes, understood only by the assembler. These do not generate machine code, but tell the assembler where the code is to be run and stored, respectively.

Although the ORG can be specified anywhere in memory, the OBJ is peculiar to older assemblers. The OBJ, or the place in memory where the code that is built is stored, must not overwrite either the assembler or the text file containing your source program.

Older assemblers, like TED II +, need to be told where the location is. Default values are recommended. Newer assemblers like BIG MAC, MERLIN, and TOOL KIT don't use OBJ pseudo-opcodes since they default to those values automatically.

When an assembler builds its code for an ORG different from its OBJ (as in the above example), the code has addresses and relative branches that will only execute at the proper ORG runtime address. The assembler, however, saves the code that is physically stored, beginning at address \$6000. It will not execute if run at that address, so that you need to load or run it at \$800 using a “A\$800” after the name of the program.

Now that you have had a taste of assembly language programming and have seen that it isn't as bad as you thought, there are a number of fundamental operations that must be learned. The most important operation is to move numbers from one memory location to another. This can be accomplished by loading a value into any one of the three internal 6502 registers, the Accumulator, X or Y registers, and storing that number somewhere in memory. A LDA (Load Accumulator) instruction can be carried out in several different ways depending on its addressing mode. First, we can load the Accumulator with a real hexadecimal value (LDA #\$05). This is called Immediate Mode Addressing. Sometimes, we need to be able to load the Accumulator with a variable stored in a memory location (LDA \$900). This is called Absolute Addressing. The only other addressing mode which we will discuss for the time being is the indexed addressing mode. It takes the form of LDA \$900,X or LDA \$900,Y depending on whether the X or Y register is used as an index. If, for example, the X register contains #\$05, then the instruction above loads the value from location $900 + 5$ or \$905. This addressing mode is used primarily for indexing into tables stored at particular memory locations.

Store operations are similar to load operations. You can store a value into an “absolute” memory location, or you can store indirectly into a memory location, offset by the value contained in either the X or Y register.

In summary, the table below shows the various load and store operations.

	ACCUMULATOR	X REGISTER	Y REGISTER
LOAD	LDA #\$05	LDX #\$05	LDY #\$05
	LDA \$900	LDX \$900	LDY \$900
	LDA \$900,X	LDX \$900,Y	
	LDA \$900,Y		LDY \$900,X
STORE	STA \$900	STX \$900	STY \$900
	STA \$900,X		STY \$900,X
	STA \$900,Y	STX \$900,Y	

Sometimes it is necessary when counting cycles or looping through code to increment or decrement a value directly - similar to a FOR-NEXT loop in BASIC. In assembly language, either the X and Y registers or any memory location can be incremented or decremented. If the X register contained \$FE, then it would contain \$FF when incremented. But if it contained \$FF, it would wrap around to become \$00. The computer informs you by setting a zero flag in its Program Status Register.

	ACCUMULATOR	X -REG	Y -REG	MEMORY LOCATION
INC BY 1	NOT AVAILABLE	INX	INY	INC \$900
DEC BY 1	NOT AVAILABLE	DEX	DEY	DEC \$900

Program flow can be altered, as in BASIC, with equivalent instructions that resemble GOTO, GOSUB, and IF-THEN statements. The JMP instruction is equivalent to a GOTO statement in that it can go to any location in the machine to continue executing code. JMP \$AD6C instructs the computer to continue executing code beginning at address \$AD6C. The GOSUB statement is identical to a JSR (Jump Subroutine) in machine language. When the computer executes the instruction JSR \$FCA8, it pushes the two-byte memory address of the instruction onto the stack, so that when it returns from the subroutine at \$FCA8 via an RTS (ReTurn from Subroutine), it will know the address of where to continue the program. When it returns, it pulls that return address off the stack and increments it by one, so that it points to the next executable instruction. The stack is like a dish dispenser. Bytes are pushed on the stack in order and pulled off in reverse order. New bytes are added to the top, while the rest of the bytes on the stack are pushed deeper.

The IF-THEN statement is simulated by a number of branch instructions which test the Program Status Register for which flags are set. Flags are usually set by compare operations. You can compare a value against the value stored in either the Accumulator or X and Y Registers. The mnemonics are CMP, CPX and CPY, respectively. For example,

```

LDA  $900  ;LOAD ACCUMULATOR WITH VALUE AT $900
CMP  #$05  ;COMPARE $5 WITH ACCUMULATOR

```

Different flags are set depending on the result.

Branch instructions are very similar to a JMP instruction (which is an unconditional branch), except that only under certain circumstances will it cause program flow to continue at a different location. For example, if we were to test for that wrap-a-round case when we incremented the X- register that contained \$FF, we would want to test the Zero Flag with a Branch Equal Zero (BEQ) instruction, and go to some label if the condition is true.

```

          LDX  $900  ;LOAD X REGISTER WITH VALUE IN MEMORY
          INX                ;INCREMENT X- REGISTER
          BEQ  SKIP  ;TEST IF 0, AND IF TRUE GO TO SKIP
          RTS                ;RETURN TO MAIN PROGRAM
SKIP      LDA  #$05
          .
          .

```

This short example loads a value from the memory location into the X register, then increments it. If wrap-a-round occurs, the test for a zero flag causes the program to jump to a label called SKIP, and the code does not return to the program that called it via the RTS. There are numerous tests on each of the flags in the Program Status Register. A summary is shown below.

BCC -	Branch if the carry flag is clear.	C = 0
BCS -	Branch if the carry flag is set.	C = 1
BEQ -	Branch if the zero flag is set	Z = 1
BNE -	Branch if the zero flag is clear	Z = 0
BMI -	Branch if minus	N = 1
BPL -	Branch if plus	N = 0
BVS -	Branch if overflow is set	V = 1
BVC -	Branch if overflow is clear	V = 0

Most assemblers offer alternative mnemonics for BCC and BCS. Since, during comparisons, the carry flag is set when the value is equal or greater than the value compared, BCS might be called BGE (Branch Greater or Equal). Likewise, BCC is equivalent to BLT (Branch Less Than). Why use these alternatives? Because they are easier to remember and visualize, and they make it clear that you are doing logical comparisons rather than testing the results of an addition or subtraction.

There is one other important concept that should be understood when doing comparisons. I implied that the subsequent branch was like a GOTO in BASIC or like a JMP instruction in machine language. This is not entirely true, since the range of the branch can not exceed -126 to +129 bytes. This is because the branch instruction is only two bytes long. The first byte is the instruction code and the second the relative address. It takes a two byte address to branch to any place in memory (Except Page Zero). The JMP instruction has the advantage that it is three bytes long. In most cases, this limitation will not cause problems. But if a branch out of range error occurs, you must reverse the test so that it will reach the required destination via a JMP instruction.

EXAMPLE: If BEQ SKIP is out of range then substitute the following:

BNE *+\$5	or	BNE A
JMP SKIP		JMP SKIP
.		A NOP
.		.

This change causes the program to drop through to the JMP instruction if the zero flag was set, and then jump to location SKIP. However, if the zero flag is not set, it will advance ahead five bytes to the instruction following the JMP. All of the other branch instructions work in a similar manner. This gives the equivalent of a Long Branch.

Simple addition and subtraction of unsigned numbers is easily accomplished in machine language. All addition and subtraction must be performed one byte at a time. Thus, large numbers or multi-byte numbers (those that exceed \$FF), must be added or subtracted one byte at a time, and the carry flag must be accounted for. It's actually not much different than addition of two multi-digit long decimal numbers. Those numbers have a digit in the one's column, another in the ten's, etc. If you add 65 to 78, you add the one's column first. Five plus eight equals 13. The value in the one's column is 3; you then carry the one into the tens digit before you add the two numbers in the ten's column. Hexadecimal addition is similar. You clear the carry before you add. If the sum of the two values exceeds \$FF, the carry is set. Since you don't clear the carry when adding the next higher byte, the resultant answer will be the sum plus the previously computed carry, as in the following example:

EXAMPLE :	+CARRY	
	63	F4
	+ 02	+ 16
	---	---
	66	0A ; SETS CARRY

The code for additions and subtractions is as follows:

ADDITIONS

```
CLC          ; CLEAR CARRY
LDA  #$F4    ; LOAD LO ORDER BYTE
ADC  #$16    ; ADD WITH CARRY
STA  LOW     ; STORE LO BYTE
LDA  #$63    ; LOAD HI ORDER BYTE
ADC  #$02    ; ADD WITH CARRY (NOTE DON'T CLEAR CARRY)
STA  HIGH    ; STORE HI BYTE
```

SUBTRACTIONS

```
SEC          ; SET CARRY FLAG
LDA  #$F4    ; LOAD VALUE
SBC  #$16    ; SUBTRACT WITH CARRY
STA  VALUE   ; STORE ANSWER
```

You should be aware that the rules for subtraction are different than for addition. The carry must be set first. This is equivalent to a borrow in subtraction. After the subtraction operation, the carry will be clear if an underflow (borrow) occurred. The carry will be set otherwise. Setting the carry is very important, a step that many beginners forget. The results are invariably incorrect if this step is skipped - and possibly even “random”, since the status of the carry flag can be on or off when the subtraction operation is performed. This can make debugging difficult.

LO-RES SCREEN

The Lo-Res screen occupies the same memory locations as the text page: \$400 to \$7FF for page one and \$800 to \$BFF for page two. When the Lo-Res graphics mode is toggled, the 1024 memory locations are presented as colored blocks rather than ASCII characters. Each ASCII character becomes two colored blocks, stacked one upon the other. Since the text page contains 24 lines of forty characters, the Lo-Res screen shows 48 rows of blocks, 40 blocks wide. Each block can be any one of 16 colors.

LOW - RESOLUTION GRAPHICS COLORS

DECIMAL	HEX	COLOR	DECIMAL	HEX	COLOR
0	\$0	BLACK	8	\$8	BROWN
1	\$1	MAGENTA	9	\$9	ORANGE
2	\$2	DARK BLUE	10	\$A	GREY II
3	\$3	PURPLE	11	\$B	PINK
4	\$4	DARK GREEN	12	\$C	LIGHT GREEN
5	\$5	GREY I	13	\$D	YELLOW
6	\$6	MEDIUM BLUE	14	\$E	AQUAMARINE
7	\$7	LIGHT BLUE	15	\$F	WHITE

Since each screen memory location represents two colored blocks in Lo-Res, each byte is divided into two equal halves called nibbles (4 bits). The value which is in the lower nibble of the byte determines the color for the upper block, and the higher order nibble determines the color for the lower block. Thus, if memory location \$400, which is the first position in the first row, contains \$D1, then the upper block is magenta and the lower block is yellow.



I would like to point out that the map of the text screen is not sequential in memory. Like its big brother, the Hi-Res screen, the first 40 bytes map across the first row, but the second 40 bytes represent a row which is a third of the way down the screen. The third 40 bytes constitute a row in the bottom third of the screen. The exact order is not important at this time, because monitor subroutines calculate the base address for any Lo-Res color plotting automatically. To plot any Lo-Res point you need only give the monitor subroutine located at \$F800 the row and column to plot and the proper color. The column is loaded into the Y register, the color into memory location \$30, and the row into the Accumulator. A call to \$F800 will plot a Lo-Res dot to the

screen, and will be seen if the Lo-Res graphics display is activated first. The dot's value is always placed into Lo-Res memory by this subroutine, even if you are viewing Hi-Res screen memory.

I would like to interject a word of caution when inputting color values for Lo-Res plotting subroutines. Because setting the proper color nibble depends on whether you are plotting on an odd or even row, it is safer to put the color desired in both low and high nibbles. To illustrate the point, let's assume we placed a \$01 in the color register and we wanted to plot the point on row 0, column 0. The plotting subroutine would use the lower order nibble \$1 to plot the magenta dot, then it would ignore the higher order nibble. However, if we choose instead to plot at row 1, column 0, the subroutine will use \$0 for the color and ignore the lo order nibble. Thus, the screen would remain black. The solution is to put the color in both nibbles. Placing \$11 in the color register will always plot the proper color in the above example anywhere on the Lo-Res screen.

	FUNCTION	Y REG	ACC.	\$0030	\$002C	\$002D
\$FC58	CLEAR SCREEN	--	--	--	--	--
\$FB40	SET GRAPHICS	--	--	--	--	--
\$F800	PLOT A POINT	COLUMN	ROW	COLOR	--	--
\$F819	HORIZ. LINE	START COLUMN	ROW COLUMN	COLOR	END	--
\$F828	VERT. LINE		START	COLOR	--	END
			ROW			ROW
\$F871	SCRN (X,Y)	COLUMN	ROW *	--	--	--

*(NOTE: COLOR RETURNED IN ACC.)

It is time to get your feet wet; we're going to plot your first few dots and lines on the Lo-Res screen. The code that I'll present is written on the TED II + assembler. However, the code is simple enough to type in on the mini-assembler if you haven't purchased an assembler as yet.

```

ORG $6000 ;ASSEMBLE CODE AT $6000
OBJ $6000
JSR $FB40 ;SET LO-RES GRAPHICS MODE
JSR $FC58 ;CLEAR SCREEN
LDA #$66 ;SET COLOR BLUE
STA $30 ;STORE IN COLOR LOCATION
LDY #$05 ;COLUMN
LDA #$03 ;ROW
JSR $F800 ;PLOT POINT
LDA #$99 ;SET COLOR ORANGE
STA $30 ;STORE IN COLOR LOCATION
LDA #$08 ;END COLUMN
STA $2C ;STORE END COLUMN
LDY #$02 ;START COLUMN
LDA #$06 ;ROW
JSR $F819 ;PLOT HORIZ ROW
RTS ;RETURN TO MONITOR

```

The above program plots a blue dot at location X = 5, Y = 3. It then draws a horizontal orange line from X = 2 ,Y = 6 to X = 8 ,Y = 6. The program can be run by typing a 6000G <CR> from the monitor. If the ORG is assembled elsewhere with another assembler type, the appropriate start. For example, if LISA assembles your code at \$800, then type 800G <CR>.

As you can see, plotting with Lo-Res graphics is relatively easy but involves tedious details. The same code in BASIC, as listed below, would have taken a mere five statements. Yet the machine language program will run at least twenty times faster.

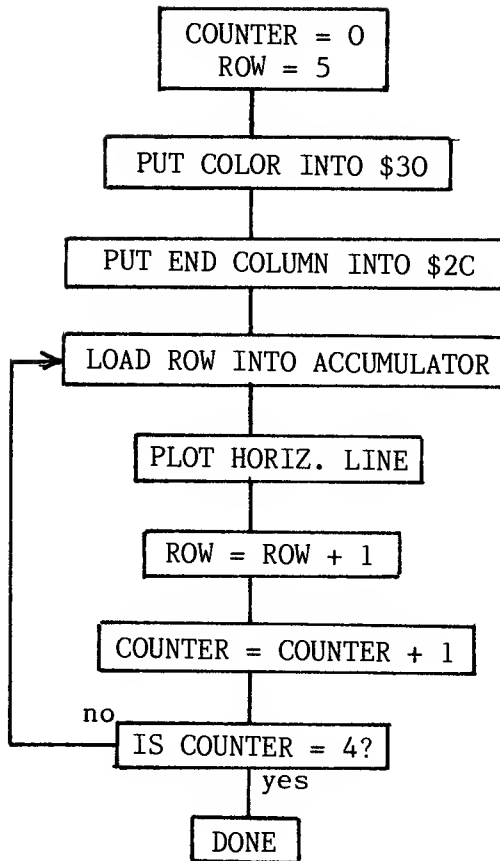
```

10 GR: COLOR = 6:PLOT 5,3
20 COLOR = 9:HLIN 2,8 at 6
30 END

```

The ability to plot several horizontal lines having the same color is useful in setting up our "Breakout" game. The code is also instructive in that it simulates the FOR-NEXT loop in BASIC. We will need a counter which we will appropriately call COUNTER. We will first initialize COUNTER to zero. Since we aren't going to begin plotting our horizontal lines at row zero but instead at row five, we will use a variable called ROW to keep track of our vertical row position. The object is to plot four horizontal red lines beginning at row 5 and extending through row 8. The beginning column for each row is \$5 and the ending column is \$22.

As we plot each row successively, we increment our variables, COUNTER and ROW. The variable COUNTER is then tested to see if it has reached the value #\$04. If it has, the code exits the loop. Otherwise, it branches back to LOOPA so that it plots the next row. When it has plotted all four red lines, it exits. The code and flow chart are shown below.



```

LDA  #$00
STA  COUNTER
LDA  #$05      ;START FIFTH ROW
STA  ROW
LDA  #$11      ;RED COLOR FIRST 4 ROWS
STA  $30      ;COLOR STORAGE
LDA  #$22      ;END COLUMN
STA  $2C
LOOPA LDA  ROW
LDY  #$05      ;START COLUMN
JSR  $F819     ;PLOT HORIZ LINE
INC  ROW      ;NEXT ROW
INC  COUNTER   ;COUNTER = COUNTER + 1
LDA  COUNTER
CMP  #$04     ;HAVE WE DONE ALL FOUR ROWS
BNE  LOOPA    ;NO! GOTO LOOPA
RTS          ;DONE!

```

The “Breakout” game involves the simplest animation technique available on the Apple. We have a ball or, in Lo-Res graphics, a dot, that bounces around the screen. It will ricochet off a moveable paddle, the walls, or any of the two-by-two sized color bricks. Movement is accomplished by erasing the ball at its old position and redrawing it at its new position. The ball is very predictable. It changes direction only upon collision, and in all cases (except contact with the paddle), simply reverses its direction. The position of contact with the paddle determines the ball’s direction. Balls striking the left end travel upwards and to the left at a 45 degree angle, while balls striking the inside left travel in the same direction but at a 60 degree angle. Balls striking the paddle’s right side travel at similar angles but to the right.

Determining where the ball struck the paddle is easy. The four block-wide paddle is always drawn at row 35 decimal or \$23, and the first block begins at PADX, a variable controlled by the paddle. The ball’s position is always at BX,BY, and it has a velocity VX,VY. By comparing the ball’s vertical position to PADX first, and then PADX + 1, etc, when a collision is detected, the ball’s velocity components VX and VY are reset. VY is always reset to -1 so that the ball travels upwards. However, VX varies with which block was hit. As we mentioned earlier, the two outside blocks would cause the ball to travel at 45 degree angles. This would mean a VX of +1 or -1. The inside blocks would cause the ball to bounce at 60 degree angles or VX at +1/2 or -1/2.

Incrementing the ball’s position by 1/2 is not possible in machine code. But if the incremented value was first doubled before calculating the ball’s new position, and the result divided by two, the same result would be obtained with the loss of the fractional part. This doesn’t matter since the ball can only be placed at whole number positions.

For example: $BX = 6$ and $VY = 1/2$

$$BX = BX + VY = 6 + 1/2 = 6 \text{ (ROUNDED).}$$

If the numbers were doubled and the result divided by two, then

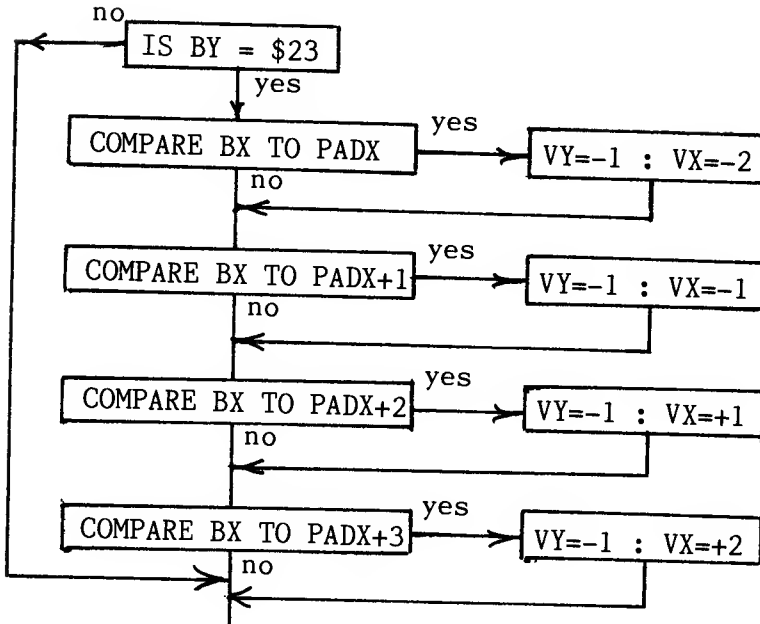
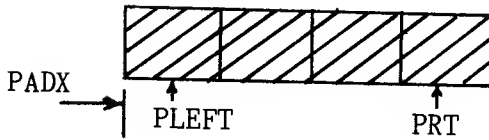
$$BX = 12 + 1 = 13/2 = 6 \text{ (ROUNDED).}$$

If the doubled position is kept rather than discarded and we wished to move the ball another 1/2 position, then

$$BX = 13 + 1 = 14/2 = 7.$$

This would result in the ball moving in the X direction every other cycle. With $VY = -1$, it would travel at a 60 degree angle upwards and towards the right.

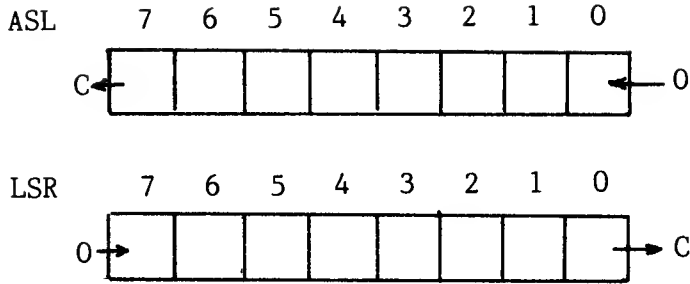
PADDLE DEFLECTOR



*Note all VX values doubled.

Multiplication and division by powers of two is easy in machine language. The mnemonic ASL is used for multiplication by two. The Arithmetic Shift Left (ASL) instruction shifts all of the bits in the Accumulator one position to the left. Thus, bit 0 is shifted into bit 1, bit 1 into bit 2, etc. Bit seven is shifted into the carry bit so that you can use the BCC and BCS instructions to test for overflows. For example, if only bit two was on (4 decimal) and we did an ASL, the bit would be shifted to bit three (8 decimal). Thus, it is easy to multiply by powers of two by doing repeated ASL instructions.

Conversely, division is performed by the Logical Shift Right (LSR) instruction. Bits are shifted to the right and the bit 0 is shifted into the carry. This is equivalent to dividing by two with loss of the fractional part.



```
LDA  #$05 ;LOAD ACCUMULATOR WITH 5
LSR          ;DIVIDE NUMBER BY TWO
STA  $900   ;VALUE STORED IN $900 IS 2
```

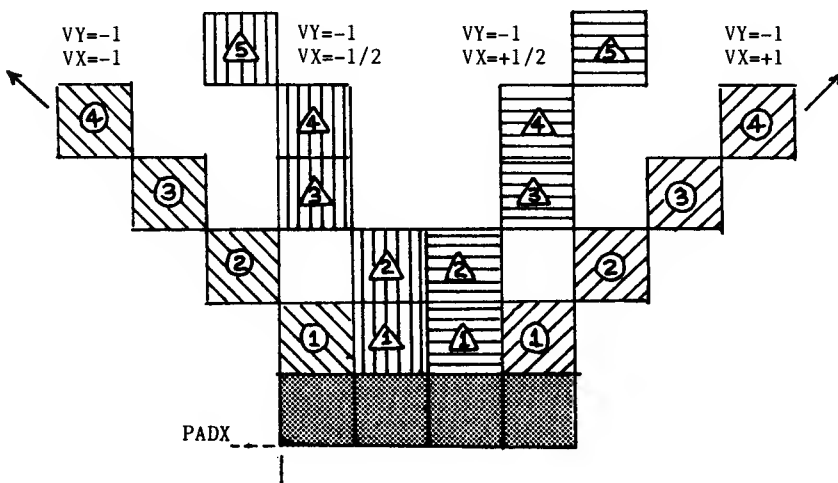
In order to update the ball's position, we take the ball's old BX,BY position in each direction and add the change in position or its directional velocity. Negative values are converted to their two's complement equivalent so that all operations are simple additions. A negative one becomes a \$FF, so that \$FF plus \$02 = \$01.

NEW POSITION = OLD POSITION + CHANGE IN POSITION

```
BX = BX + VX    X DIRECTION
BY = BY + VY    Y DIRECTION
```

The ball's X position is calculated using doubled position values DBX and doubled velocities values VX to avoid 1/2 values

Thus, $DBX = DBX + VX$ and $BX = DBX/2$.



```

LDA  DBX  ;OLD DOUBLED X POSITION
CLC
ADC  VX   ;X DIRECTION VALUE
STA  DBX  ;THIS DOUBLED VALUE WILL RETAIN FRACTION
LSR           ;DIVIDE BY 2 , WILL LOSE FRACTION
STA  BX   ;NEW BALL X POSITION
LDA  BY   ;OLD Y POSITION OF BALL
CLC
ADC  VY   ;ADD Y DIRECTION VELOCITY
STA  BY   ;NEW BALL Y POSITION

```

As the ball bounces around the screen, it will soon collide with one of the colored 2 by 2 bricks at the top of the screen. Since these are colored blocks, collisions can be detected between the ball and these blocks with the SCRNB function. This monitor subroutine will return the value of the color at any position. This test is performed before the ball is drawn to the screen, or the test becomes meaningless at the ball's position since the ball will plot over the background color blocks.

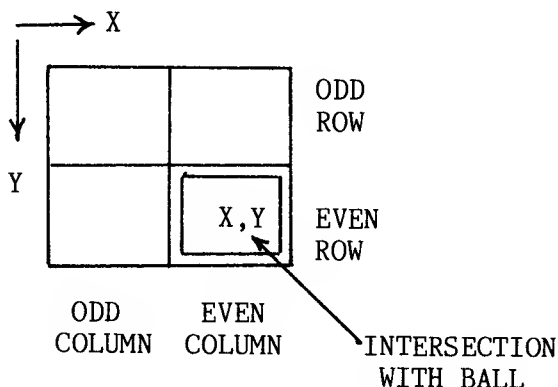
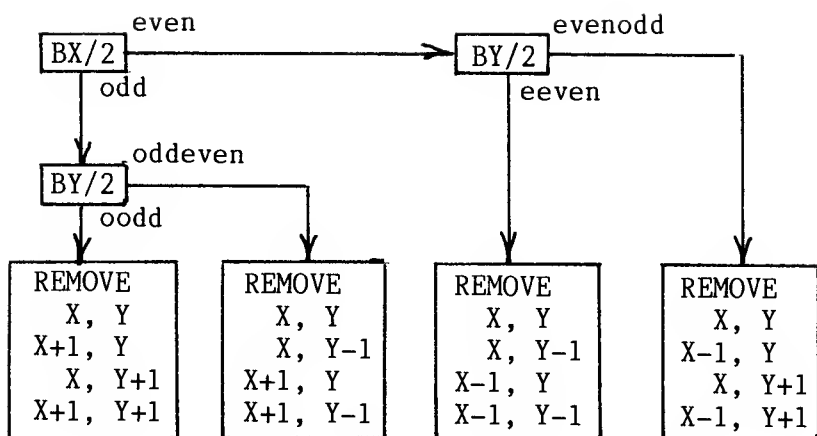
We will want to delete the block if a non-black (background) color is returned during the test. The brick is four times larger than our ball, so we must delete all four blocks at once. This is a troublesome operation, since we might have collided with any of the four color blocks that comprise the brick. The block that we hit is BX,BY. If we hit the top left block of the brick we will want to delete block BX,BY ,BX +1,BY , BX +1,BY +1 , and BX,BY +1. The other three possible collisions with the brick have completely different sequences of blocks to be removed.

Bricks always begin in an odd row, at an odd column. A test can be made to see if our ball is in an odd or even row, or an odd or even column. That will determine which of four sequences of blocks to remove. An odd even test can be done on BX using a division by two or LSR instruction. Odd values always have a one in the bit zero position. An LSR operation shifts them to the carry bit. Therefore, odd values set the carry. A BCC (Branch Carry Clear) test will determine if the value is odd or even.

```

      LDA  BX
      LSR           ;DIVIDE BY TWO
      BCC  EVEN  ;BX IS EVEN IF CARRY IS CLEAR
ODD   JMP  SKIP
EVEN  NOP           ;CONTINUE WITH EVEN CODE

```



Once the block is removed, the score must be incremented by the point value for each block. In this game, yellow is worth one point, blue two points, and red three points. The score is kept in a memory location called SUM. There has been no attempt in this example to convert the hexadecimal value of SUM to a decimal value. That type of scorekeeping routine is outlined in Chapter 6.

The scorekeeping routine first checks the color of the block hit for yellow. If it is equal to `#$0D` (Yellow) it will add `#$01` to SUM. Otherwise, it will branch to the label NEXT. There it encounters a test for the color blue. If the block isn't blue it branches to the label NEXT1. If it is blue, `#$02` is added to SUM, otherwise `#$03` is added to SUM because it must be red.


```

SCORE    LDA    COLOR
          CMP    #$0D          ;HIT YELLOW?
          BNE    NEXT
          LDA    SUM
          CLC
          ADC    #$01
          STA    SUM
          JMP    SCORE1
NEXT      LDA    COLOR
          CMP    #$06          ;HIT BLUE?
          BNE    NEXT1
          LDA    SUM
          CLC
          ADC    #$02
          STA    SUM
          JMP    SCORE1
NEXT1     LDA    COLOR
          CMP    #$01          ;HIT RED?
          BNE    SCORE1
          LDA    SUM
          CLC
          ADC    #$03
          STA    SUM
SCORE1    JSR    PRINT
          CMP    #$FO          ;SUM=240 FOR ALL BLOCKS
          BGE    END

```

This score will be printed in the text window below the Lo-Res graphics. We want to print the letters SCORE followed by the value in SUM. There is a monitor subroutine called COUT that outputs a single character to the screen. If the cursor position has been previously set, any ASCII character placed into the Accumulator will be outputted to the screen. Since strings are usually more than one character, the code must be looped so that each character is retrieved in its turn, then placed on the screen by COUT. The string can be stored as a hexadecimal table in memory beginning at a location labeled STRING. Each time we load the Accumulator, we index into the table X bytes where X is the value in the X-Register. They call the operation LDA STRING, X ,Indirect Addressing. The X-Register begins at #\$00 and is incremented after each byte is outputted to the screen.

A test is needed to detect the end of the string. Since a general purpose print output routine is desired for any length string up to 255 characters , it is best not to restrict the test to detecting the length of the string, but to detect a character that is never sent to the screen. The hexadecimal 00 (the reverse @ sign) is rarely used and is a good choice for a test byte. When the code detects

this byte, it knows it has completed the string and exits the print loop. The value of SUM is then outputted by the monitor subroutine PRBYTE, which prints a single hexadecimal byte. The print subroutine is shown below.

```

PRINT  LDX  #$00          ;INDEX INTO STRING BEGINS AT 0
        LDA  #$05
        STA  $24          ;HTAB5
        LDA  #$17
        JSR  TABV         ;VTAB23
PRINT1  LDA  STRING,X     ;GET Xth ELEMENT OF STRING
        BEQ  DONE        ;FINISHED?
        JSR  COUT         ;PRINT LETTER
        INX             ;NEXT ELEMENT
        JMP  PRINT1      ;LOOP
DONE    LDA  SUM
        JSR  PRBYTE      ;OUTPUT BYTE SUM
        RTS
STRING  ASC  "SCORE = "
        HEX  00

```

The “Breakout” game needs paddle control. The paddle is used both to initially start the game by a button press, and to move the deflector back and forth at the bottom of the screen. Button presses are the easiest to detect. There are three paddle switches that are located at \$C061 – \$C063. The lowest hardware location is for paddle #0. If the button is pushed, the value loaded into the Accumulator is negative. The program can be put into an endless loop waiting for a button press with the following code:

```

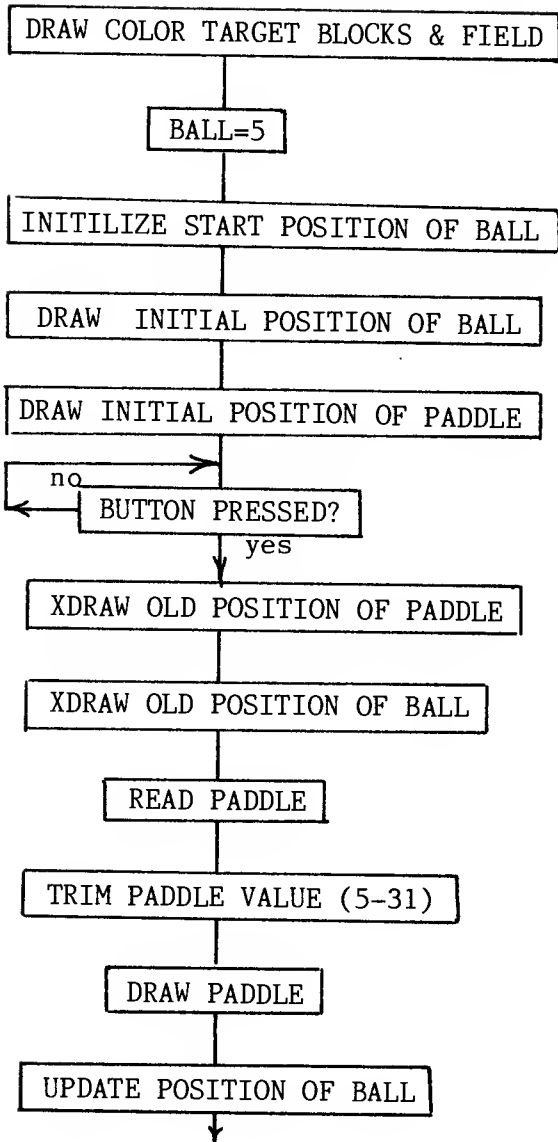
BUTTON  LDA  $C061
        BPL  BUTTON

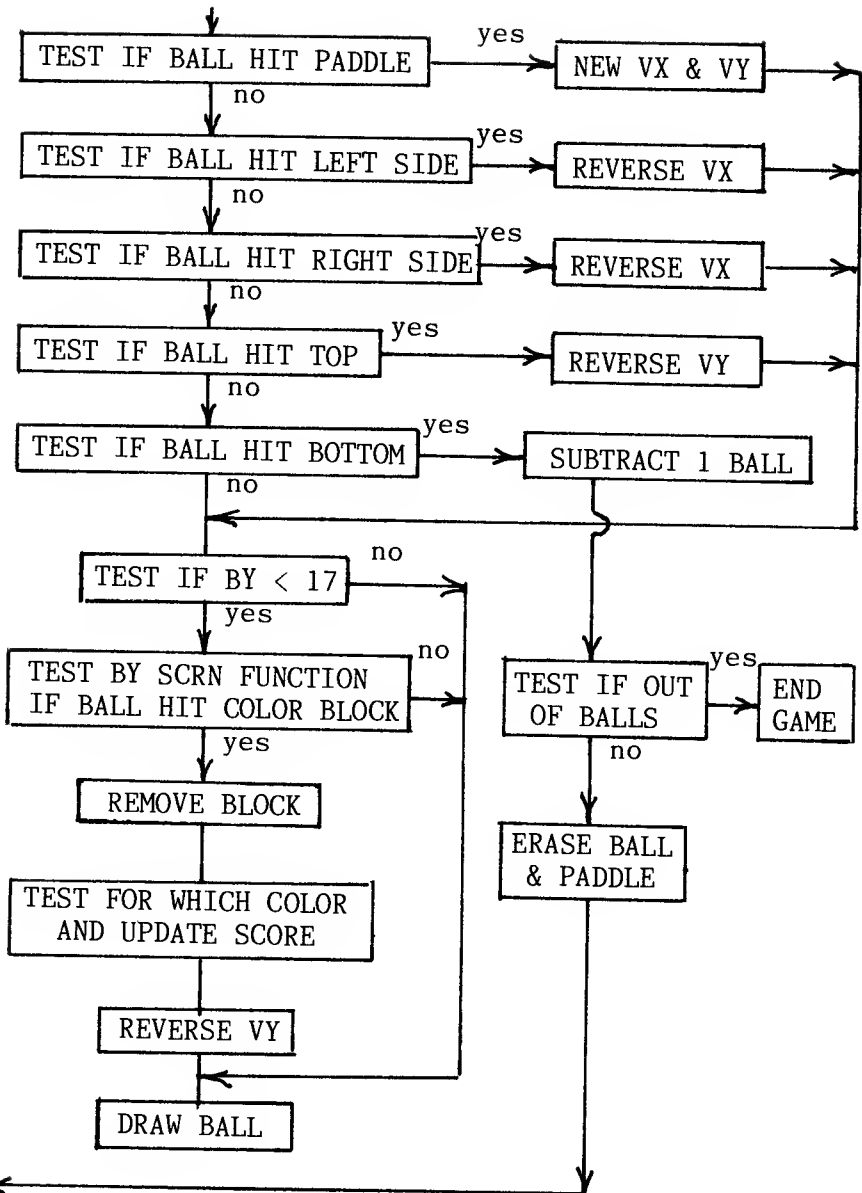
```

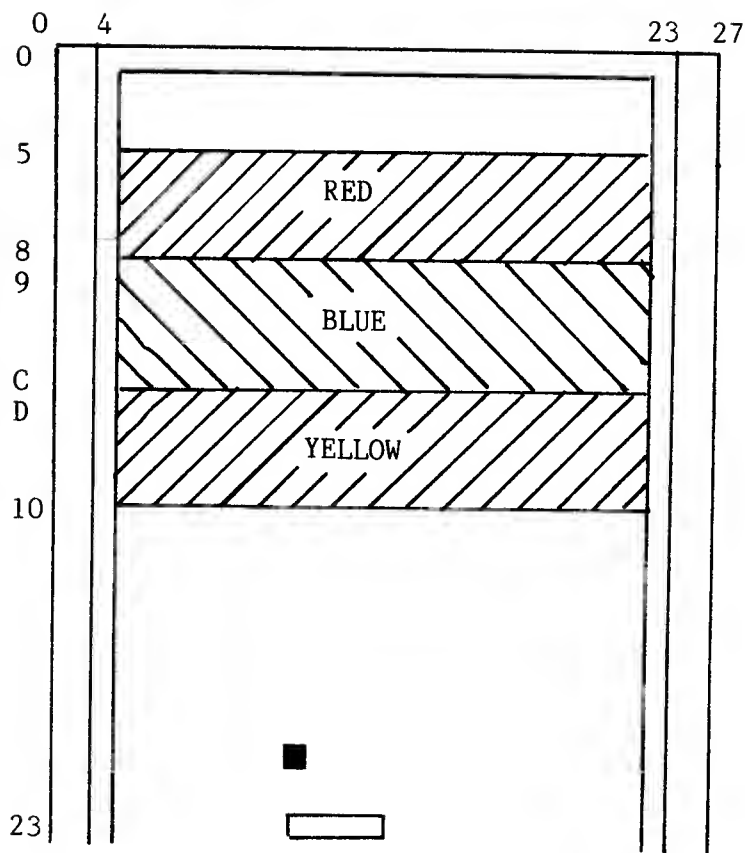
The code will only exit the loop if the button is pressed.

The paddle’s output value (0-255) can be read by accessing a monitor subroutine called PREAD, located at \$FB1E. The paddle number is placed into the X-Register and the value of the paddle is outputted to the Y-Register. It is directly equivalent to the BASIC command PDL(0). In our case, we need the output clipped to a value (0-31). It is first necessary to divide the value by four. This gives a value between 0-64. This range was chosen rather than 0-32, so that the player has better control with half the amount of paddle turning. The value is then tested to be within that range. If it is less than \$05 it is set to \$05, and if greater than \$1F (decimal 31), it is set equal to \$1F. This is called clipping.

We have covered all of the pertinent code that is necessary to write a “Breakout” game. The only thing left is the flowchart, and that is shown below. The complete assembled code follows.







BREAKOUT SCREEN

```

1      ** B R E A K O U T   G A M E **
2      ORG   $6000
6000: 4C 17 60 3      JMP   PROG           ;JMP TO MAIN PROGRAM
4      ROW    DS    1
5      COUNTER DS    1
6      BX     DS    1
7      BY     DS    1
8      BBX    DS    1
9      BBY    DS    1
10     VX     DS    1
11     VY     DS    1
12     DBX    DS    1
13     PDX    DS    1

```

	14	PADX	DS	1	
	15	PRT	DS	1	
	16	PLEFT	DS	1	
	17	SUM	DS	1	
	18	BALL	DS	1	
	19	COLOR	DS	1	
	20	CBALL	DS	1	
	21	CPDL	DS	1	
	22	PITCH	DS	1	
	23	TIME	DS	1	
	24	PREAD	EQU	\$FB1E	
	25	COUT	EQU	\$FDFO	
	26	TABV	EQU	\$FB5B	
	27	PRBYTE	EQU	\$FDDA	
6017:	20 40 FB	28	PROG	JSR	\$FB40 ;SET LORES GRAPHICS MODE
601A:	20 58 FC	29		JSR	\$FC58 ;CLEAR SCREEN
		30	*DRAW SCREEN & BLOCKS		
601D:	A9 88	31	LDA	#\$88	;SET COLOR BROWN
601F:	85 30	32	STA	\$30	
6021:	A9 23	33	LDA	#\$23	;END COLUMN
6023:	85 2C	34	STA	\$2C	
6025:	A9 00	35	LDA	#\$00	;TOP ROW
6027:	A0 04	36	LDY	#\$04	;START COLUMN
6029:	20 19 F8	37	JSR	\$F819	;PLOT HORIZ LINE
602C:	A9 27	38	LDA	#\$27	;END ROW
602E:	85 2D	39	STA	\$2D	
6030:	A9 01	40	LDA	#\$01	;START ROW
6032:	A0 04	41	LDY	#\$04	;COLUMN
6034:	20 28 F8	42	JSR	\$F828	;PLOT VERT LINE
6037:	A9 01	43	LDA	#\$01	;START ROW
6039:	A0 23	44	LDY	#\$23	;COLUMN
603B:	20 28 F8	45	JSR	\$F828	;PLOT VERT LINE
603E:	A9 00	46	LDA	#\$00	
6040:	8D 04 60	47	STA	COUNTER	
6043:	A9 05	48	LDA	#\$05	;START 5TH ROW
6045:	8D 03 60	49	STA	ROW	
6048:	A9 11	50	LDA	#\$11	;RED COLOR FIRST 4 ROWS
604A:	85 30	51	STA	\$30	
604C:	A9 22	52	LDA	#\$22	;END COLUMN
604E:	85 2C	53	STA	\$2C	
6050:	AD 03 60	54	LOOPA	LDA	ROW
6053:	A0 05	55	LDY	#\$05	;START COLUMN
6055:	20 19 F8	56	JSR	\$F819	;PLOT HORIZ LINE
6058:	EE 03 60	57	INC	ROW	;NEXT ROW
605B:	EE 04 60	58	INC	COUNTER	
605E:	AD 04 60	59	LDA	COUNTER	
6061:	C9 04	60	CMP	#\$04	
6063:	D0 EB	61	BNE	LOOPA	
6065:	A9 66	62	LDA	#\$66	;BLUE COLOR NEXT 4 ROWS
6067:	85 30	63	STA	\$30	
6069:	AD 03 60	64	LOOPB	LDA	ROW
606C:	A0 05	65	LDY	#\$05	;START COLUMN
606E:	20 19 F8	66	JSR	\$F819	;PLOT HORIZ LINE
6071:	EE 03 60	67	INC	ROW	
6074:	EE 04 60	68	INC	COUNTER	
6077:	AD 04 60	69	LDA	COUNTER	
607A:	C9 08	70	CMP	#\$08	
607C:	D0 EB	71	BNE	LOOPB	
607E:	A9 DD	72	LDA	#\$DD	;YELLOW COLOR
6080:	85 30	73	STA	\$30	

```

6082: AD 03 60 74   LOOPC   LDA   ROW
6085: A0 05 75      LDY   #$05           ;START COLUMN
6087: 20 19 F8 76      JSR   $F819
608A: EE 03 60 77      INC   ROW
608D: EE 04 60 78      INC   COUNTER
6090: AD 04 60 79      LDA   COUNTER
6093: C9 0C 80      CMP   #SOC
6095: D0 EB 81      BNE   LOOPC
6097: A9 05 82      LDA   #$05
6099: 8D 11 60 83      STA   BALL
609C: A9 00 84      LDA   #00
609E: 8D 10 60 85      STA   SUM
      86   *INITIALIZE VARIABLES
60A1: A9 14 87   START   LDA   #$14           ;INITIAL POSITION BALL
60A3: 8D 05 60 88      STA   BX
60A6: 8D 06 60 89      STA   BY
60A9: A9 28 90      LDA   #$28
60AB: 8D 0B 60 91      STA   DBX
60AE: A9 00 92      LDA   #00           ;INITIAL VELOCITY BALL
60B0: 8D 09 60 93      STA   VX
60B3: A9 01 94      LDA   #01
60B5: 8D 0A 60 95      STA   VY
60B8: A9 11 96      LDA   #$11           ;INITIAL PADDLE POSITION
60BA: 8D 0D 60 97      STA   PADX
60BD: A9 14 98      LDA   #$14
60BF: 8D 0E 60 99      STA   PRT
60C2: A9 FF 100     LDA   #$FF           ;WHITE BALL
60C4: 8D 13 60 101    STA   CBALL
60C7: A9 CC 102     LDA   #CC           ;GREEN PADDLE
60C9: 8D 14 60 103    STA   CPDL
      104  *PRINT INITIAL SCORE
60CC: 20 C2 63 105    JSR   PRINT
      106  *DRAW INITIAL POSITIONS BALL& PADDLE
60CF: AD 13 60 107    LDA   CBALL
60D2: 85 30 108      STA   $30
      109
60D4: AC 05 60 109    LDY   BX           ;COLUMN
60D7: AD 06 60 110    LDA   BY           ;ROW
60DA: 20 00 F8 111    JSR   $F800           ;PLOT BALL
60DD: AD 14 60 112    LDA   CPDL
60E0: 85 30 113      STA   $30
60E2: AD 0E 60 114    LDA   PRT
60E5: 85 2C 115      STA   $2C
60E7: AC 0D 60 116    LDY   PADX           ;START COLUMN
60EA: A9 23 117      LDA   #$23           ;PADDLE ROW
60EC: 20 19 F8 118    JSR   $F819           ;PLOT PADDLE
      119  *START GAME WITH BUTTON
60EF: AD 61 C0 120    BUTTON  LDA   $C061           ;NEG IF BUTTON PRESSED
60F2: 10 FB 121      BPL   BUTTON
      122  *
      123  ** M A I N   P R O G R A M   L O O P **
      124  *
      125  *XDRAW OLD POSITIONS BALL& PADDLE
60F4: A9 00 126   MAIN   LDA   #00
60F6: 85 30 127      STA   $30
60F8: AC 05 60 128    LDY   BX
60FB: AD 06 60 129    LDA   BY
60FE: 20 00 F8 130    JSR   $F800           ;XPLOT BALL
6101: AD 0E 60 131    LDA   PRT
6104: 85 2C 132      STA   $2C
6106: AC 0D 60 133    LDY   PADX

```

6109:	A9 23	134	LDA	#\$23	
610B:	20 19 F8	135	JSR	\$F819	;XPLOT PADDLE
		136	*READ	PADDLE	
610E:	A2 00	137	LDX	#\$00	;PADDLE 0
6110:	20 1E FB	138	JSR	PREAD	
6113:	98	139	TYA		;PADDLE VALUE(0-255) IN Y REG
6114:	4A	140	LSR		;DIVIDE BY 4
6115:	4A	141	LSR		
6116:	C9 20	142	CMP	#\$20	;CLIP TO (5-31)
6118:	90 05	143	BLT	SKIPP	
611A:	A9 1F	144	LDA	#\$1F	
611C:	8D 0D 60	145	STA	PADX	
611F:	C9 05	146	SKIPP	CMP	#\$05
6121:	B0 02	147	BGE	SKIPP1	
6123:	A9 05	148	LDA	#\$05	
6125:	8D 0D 60	149	SKIPP1	STA	PADX
6128:	18	150	CLC		
6129:	69 03	151	ADC	#\$03	
612B:	8D 0E 60	152	STA	PRT	
		153	*DRAW	NEW POSITION PADDLE	
612E:	AD 14 60	154	LDA	CPDL	
6131:	85 30	155	STA	\$30	
6133:	AD 0E 60	156	LDA	PRT	
6136:	85 2C	157	STA	\$2C	
6138:	AC 0D 60	158	LDY	PADX	
613B:	A9 23	159	LDA	#\$23	;ROW
613D:	20 19 F8	160	JSR	\$F819	; PLOT HORIZ PADDLE
		161	*UPDATE	POSITION BALL	
		162	*NOTE	ALL VX VALUES DOUBLED TO AVOID 1/2 VALUES	
6140:	AD 0B 60	163	LDA	DBX	;OLD DOUBLED X POS VALUE
6143:	18	164	CLC		
6144:	6D 09 60	165	ADC	VX	;X DIRECTION VELOCITY
6147:	8D 0B 60	166	STA	DBX	;THIS DOUBLED VALUE WILL KEEP FRACT-
		167	*~		TIONAL PART OF NEW POSITION
614A:	4A	168	LSR		;HALF VALUE WILL LOSE FRACTION
614B:	8D 05 60	169	STA	BX	;NEW BALL X POS
614E:	AD 06 60	170	LDA	BY	;OLD Y POS
6151:	18	171	CLC		
6152:	6D 0A 60	172	ADC	VY	;ADD Y DIRECTION VELOCITY
6155:	8D 06 60	173	STA	BY	;NEW BALL Y POSITION
		174	*TEST IF	BALL HIT SIDES OR PADDLE	
6158:	AD 06 60	175	PADDLE	LDA	BY
615B:	C9 23	176	CMP	#\$23	;AT PADDLE ROW?
615D:	FO 03	177	BEQ	PAD1	;YES!
615F:	4C B7 61	178	JMP	LEFT	
6162:	AD 0D 60	179	PAD1	LDA	PADX
6165:	8D 0F 60	180	STA	PLEFT	
6168:	AD 05 60	181	FIRST	LDA	BX
616B:	CD 0F 60	182	CMP	PLEFT	
616E:	DO 0A	183	BNE	SECOND	
6170:	A9 FF	184	LDA	#\$FF	
6172:	8D 0A 60	185	STA	VY	;VY=-1
6175:	A9 FE	186	LDA	#\$FE	
6177:	8D 09 60	187	STA	VX	;VX=-2
617A:	EE 0F 60	188	SECOND	INC	PLEFT
617D:	AD 05 60	189	LDA	BX	
6180:	CD 0F 60	190	CMP	PLEFT	
6183:	DO 08	191	BNE	THIRD	
6185:	A9 FF	192	LDA	#\$FF	
6187:	8D 0A 60	193	STA	VY	;VY=-1

618A:	8D 09 60 194		STA VX	;VX=-1
618D:	EE 0F 60 195	THIRD	INC PLEFT	
6190:	AD 05 60 196		LDA BX	
6193:	CD 0F 60 197		CMP PLEFT	
6196:	DO 0A 198		BNE FOURTH	
6198:	A9 FF 199		LDA #\$FF	
619A:	8D 0A 60 200		STA VY	;VY=-1
619D:	A9 01 201		LDA #\$01	
619F:	8D 09 60 202		STA VX	;VX=1
61A2:	EE 0F 60 203	FOURTH	INC PLEFT	
61A5:	AD 05 60 204		LDA BX	
61A8:	CD 0F 60 205		CMP PLEFT	
61AB:	DO 0A 206		BNE LEFT	
61AD:	A9 FF 207		LDA #\$FF	
61AF:	8D 0A 60 208		STA VY	;VY=-1
61B2:	A9 02 209		LDA #\$02	
61B4:	8D 09 60 210		STA VX	;VX=2
61B7:	AD 05 60 211	LEFT	LDA BX	
61BA:	C9 06 212		CMP #\$06	;HIT LEFT SIDE?
61BC:	BO 0B 213		BGE RIGHT	;NO!
61BE:	AD 09 60 214		LDA VX	;REVERSE VX
61C1:	49 FF 215		EOR #\$FF	;COMPLEMENT
61C3:	8D 09 60 216		STA VX	
61C6:	EE 09 60 217		INC VX	;VALUE CORRECTED
61C9:	AD 05 60 218	RIGHT	LDA BX	
61CC:	C9 22 219		CMP #\$22	;HIT RIGHT SIDE?
61CE:	90 0B 220		BLT TOP	;NO!
61D0:	AD 09 60 221		LDA VX	;REVERSE VX
61D3:	49 FF 222		EOR #\$FF	;COMPLEMENT
61D5:	8D 09 60 223		STA VX	
618:	EE 09 60 224		INC VX	;VALUE CORRECTED
61DB:	AD 06 60 225	TOP	LDA BY	
61DE:	C9 01 226		CMP #\$01	;HIT TOP?
61E0:	DO 0B 227		BNE BOTTOM	;NO!
61E2:	AD 0A 60 228		LDA VY	;REVERSE VY
61E5:	49 FF 229		EOR #\$FF	;COMPLEMENT
61E7:	8D 0A 60 230		STA VY	
61EA:	EE 0A 60 231		INC VY	;VALUE CORRECTED
61ED:	AD 06 60 232	BOTTOM	LDA BY	
61F0:	C9 27 233		CMP #\$27	
61F2:	DO 3A 234		BNE BLOCKS	
61F4:	CE 11 60 235		DEC BALL	
61F7:	A9 FF 236		LDA #\$FF	;BAD SOUND FOR MISSING
61F9:	8D 15 60 237		STA PITCH	
61FC:	8D 16 60 238		STA TIME	
61FF:	20 E9 63 239		JSR SOUND	
6202:	A9 FF 240		LDA #\$FF	;SHORT DELAY
6204:	20 A8 FC 241		JSR \$FCA8	
6207:	AD 11 60 242		LDA BALL	
620A:	C9 00 243		CMP #\$00	;ALL BALLS GONE?
620C:	DO 03 244		BNE CONT	
620E:	4C DD 62 245		JMP END	
	246	*ERASE BALL &	PADDLE	
6211:	A9 00 247	CONT	LDA #\$00	
6213:	85 30 248		STA \$30	
6215:	AC 05 60 249		LDY BX	
6218:	AD 06 60 250		LDA BY	
621B:	20 00 F8 251		JSR \$F800	;XPLOT BALL
621E:	AD 0E 60 252		LDA PRT	

```

6221: 85 2C    253      STA  $2C
6223: AC 0D 60 254      LDY  PADX
6226: A9 23    255      LDA  #$23
6228: 20 19 F8 256      JSR  $F819      ;XPLOT PADDLE
622B: 4C A1 60 257      JMP  START
622E: AD 06 60 258 BLOCKS LDA  BY
6231: C9 11    259      CMP  #$11      ;IN AREA OF BLOCKS?
6233: 90 03    260      BLT  SK2      ;YES!
6235: 4C C7 62 261      JMP  DRAW
262 *TEST COLLISION WITH BLOCK VIA SCR N FUNCTION
6238: AC 05 60 263 SK2   LDY  BX      ;COLUMN
623B: AD 06 60 264      LDA  BY      ;ROW
623E: 20 71 F8 265      JSR  $F871      ;SCR N(X,Y)
6241: 8D 12 60 266      STA  COLOR      ;RETURNS OLOR IN ACC.
6244: C9 00    267      CMP  #$00      ;IS BLACK?
6246: D0 03    268      BNE  NBLACK
6248: 4C C7 62 269      JMP  DRAW      ;YES!
270 *FIND WHICH OF FOUR SUBBLOCKS HIT
624B: AD 05 60 271 NBLACK LDA  BX
624E: 4A      272      LSR           ;BX/2
624F: 90 12 273      BCC  EVEN
6251: AD 06 60 274 ODD   LDA  BY
6254: 4A      275      LSR           ;BY/2
6255: 90 06 276      BCC  ODDEVEN
6257: 20 DE 62 277 OODD  JSR  OODDS
625A: 4C 72 62 278      JMP  REV
625D: 20 17 63 279 ODDEVEN JSR  ODDEVENS
6260: 4C 72 62 280      JMP  REV
6263: AD 06 60 281 EVEN  LDA  BY
6266: 4A      282      LSR           ;BY/2
6267: 90 06 283      BCC  EEVEN
6269: 20 89 63 284 EVENODD JSR  EVENODDS
626C: 4C 72 62 285      JMP  REV
626F: 20 50 63 286 EEVEN JSR  EEVENS
287 *REVERSE VY
6272: AD 0A 60 288 REV   LDA  VY
6275: 49 FF 289      FOR  #$FF
6277: 8D 0A 60 290      STA  VY
627A: EE 0A 60 291      INC  VY
292 *CHECK COLOR & UPDATE SCORE
627D: AD 12 60 293 SCORE LDA  COLOR
6280: C9 0D 294      CMP  #$0D      ;HIT YELLOW?
6282: D0 0C 295      BNE  NEXT
6284: AD 10 60 296      LDA  SUM
6287: 18      297      CLC
6288: 69 01 298      ADC  #$01
628A: 8D 10 60 299      STA  SUM
628D: 4C B3 62 300      JMP  SCORE1
6290: AD 12 60 301 NEXT  LDA  COLOR
6293: C9 06 302      CMP  #$06      ;HIT BLUE?
6295: D0 0C 303      BNE  NEXT1
6297: AD 10 60 304      LDA  SUM
629A: 18      305      CLC
629B: 69 02 306      ADC  #$02
629D: 8D 10 60 307      STA  SUM
62A0: 4C B3 62 308      JMP  SCORE1
62A3: AD 12 60 309 NEXT1 LDA  COLOR
62A6: C9 01 310      CMP  #$01      ;HIT RED?
62A8: D0 09 311      BNE  SCORE1

```

```

62AA: AD 10 60 312      LDA SUM
62AD: 18                313      CLC
62AE: 69 03            314      ADC #$03
62BO: 8D 10 60 315      STA SUM
62B3: 20 C2 63 316      SCORE1 JSR PRINT
62B6: C9 F0            317      CMP #$F0          ;SUM=240 FOR ALL BLOCKS
62B8: B0 23            318      BGE END
62BA: A9 50            319      *SOUND FOR HITTING BLOCK
62BC: 8D 15 60 321      LDA #$50
62BF: A9 25            322      STA PITCH
62C1: 8D 16 60 323      LDA #$25
62C4: 20 E9 63 324      STA TIME
62C7: AD 13 60 325      JSR SOUND
62CA: 85 30            326      *DRAW BALL
62CC: AC 05 60 327      DRAW   LDA CBALL
62CF: AD 06 60 328      STA $30
62D2: 20 00 F8 330      LDY BX          ;COLUMN
62D5: A9 80            331      LDA BY          ;ROW
62D7: 20 A8 FC 333      LDA $F800        ;PLOT BALL
62DA: 4C F4 60 334      JSR $F800
62DD: 60                335      *DELAY
62D5: A9 80            336      LDA #$80
62D7: 20 A8 FC 333      JSR $FCA8          ;SHORT DELAY
62DA: 4C F4 60 334      JMP MAIN
62DD: 60                335      END
62D5: A9 80            336      RTS          ;RETURN TO MONITOR AT END OF GAME
62D7: 20 A8 FC 333      *
62DA: 4C F4 60 334      ** S U B R O U T I N E S **
62DD: 60                338      *
62D5: A9 80            339      *ERASE BLOCK SUBROUTINES
62D7: 20 A8 FC 333      *
62DA: 4C F4 60 334      *
62DE: A9 00            341      OODDS   LDA #$00
62E0: 85 30            342      STA $30          ;BLACK
62E2: AD 05 60 343      LDA BX
62E5: 8D 07 60 344      STA BBX          ;TEMP VALUE
62E8: A8                345      TAY          ;COLUMN
62E9: AD 06 60 346      LDA BY          ;ROW
62EC: 8D 08 60 347      STA BBY          ;TEMP VALUE
62EF: 20 00 F8 348      JSR $F800        ;ERASE PT X,Y
62F2: EE 07 60 349      INC BBX
62F5: AC 07 60 350      LDY BBX          ;COLUMN
62F8: AD 08 60 351      LDA BBY          ;ROW
62FB: 20 00 F8 352      JSR $F800        ;ERASE PT X+1,Y
62FE: EE 08 60 353      INC BBY
6301: AC 07 60 354      LDY BBX          ;COLUMN
6304: AD 08 60 355      LDA BBY          ;ROW
6307: 20 00 F8 356      JSR $F800        ;ERASE PT X+1,Y+1
630A: CE 07 60 357      DEC BBX
630D: AC 07 60 358      LDY BBX          ;COLUMN
6310: AD 08 60 359      LDA BBY          ;ROW
6313: 20 00 F8 360      JSR $F800        ;ERASE PT X,Y+1
6316: 60                361      RTS
6317: A9 00            362      ODDEVENS LDA #$00
6319: 85 30            363      STA $30          ;BLACK
631B: AD 05 60 364      LDA BX
631E: 8D 07 60 365      STA BBX
6321: A8                366      TAY          ;COLUMN
6322: AD 06 60 367      LDA BY          ;ROW
6325: 8D 08 60 368      STA BBY
6328: 20 00 F8 369      JSR $F800        ;ERASE PT X,Y
632B: CE 08 60 370      DEC BBY
632E: AC 07 60 371      LDY BBX          ;COLUMN

```

6331:	AD	08	60	372		LDA	BBY		;ROW
6334:	20	00	F8	373		JSR	\$F800		;ERASE PT X,Y-1
6337:	EE	07	60	374		INC	BBX		
633A:	AC	07	60	375		LDY	BBX		;COLUMN
633D:	AD	08	60	376		LDA	BBY		;ROW
6340:	20	00	F8	377		JSR	\$F800		;ERASE PT X+1,Y-1
6343:	EE	08	60	378		INC	BBY		
6346:	AC	07	60	379		LDY	BBX		;COLUMN
6349:	AD	08	60	380		LDA	BBY		;ROW
634C:	20	00	F8	381		JSR	\$F800		;ERASE PT X+1,Y
634F:	60			382		RTS			
6350:	A9	00		383	EVENNS	LDA	#\$00		
6352:	85	30		384		STA	\$30		
6354:	AD	05	60	385		LDA	BX		
6357:	8D	07	60	386		STA	BBX		
635A:	A8			387		TAY			;COLUMN
635B:	AD	06	60	388		LDA	BY		;ROW
635E:	8D	08	60	389		STA	BBY		
6361:	20	00	F8	390		JSR	\$F800		;ERASE PT X,Y
6364:	CE	08	60	391		DEC	BBY		
6367:	AC	07	60	392		LDY	BBX		;COLUMN
636A:	AD	08	60	393		LDA	BBY		;ROW
636D:	20	00	F8	394		JSR	\$F800		;ERASE PT X,Y-1
6370:	CE	07	60	395		DEC	BBX		
6373:	AC	07	60	396		LDY	BBX		;COLUMN
6376:	AD	08	60	397		LDA	BBY		;ROW
6379:	20	00	F8	398		JSR	\$F800		;ERASE PT X-1,Y-1
637C:	EE	08	60	399		INC	BBY		
637F:	AC	07	60	400		LDY	BBX		;CLUMN
6382:	AD	08	60	401		LDA	BBY		;ROW
6385:	20	00	F8	402		JSR	\$F800		;ERASE PT X-1,Y
6388:	60			403		RTS			
6389:	A9	00		404	EVENODDS	LDA	#\$00		
638B:	85	30		405		STA	\$30		
638D:	AD	05	60	406		LDA	BX		
6390:	8D	07	60	407		STA	BBX		
6393:	A8			408		TAY			;COLUMN
6394:	AD	06	60	409		LDA	BY		;ROW
6397:	8D	08	60	410		STA	BBY		
639A:	20	00	F8	411		JSR	\$F800		;ERASE PT X,Y
639D:	CE	07	60	412		DEC	BBX		
63A0:	AC	07	60	413		LDY	BBX		;COLUMN
63A3:	AD	08	60	414		LDA	BBY		;ROW
63A6:	20	00	F8	415		JSR	\$F800		;ERASE PT X-1,Y
63A9:	EE	08	60	416		INC	BBY		
63AC:	AC	07	60	417		LDY	BBX		;COLUMN
63AF:	AD	08	60	418		LDA	BBY		;ROW
63B2:	20	00	F8	419		JSR	\$F800		;ERASE PT X-1,Y+1
63B5:	EE	07	60	420		INC	BBX		
63B8:	AC	07	60	421		LDY	BBX		;COLUMN
63BB:	AD	08	60	422		LDA	BBY		;ROW
63BE:	20	00	F8	423		JSR	\$F800		;ERASE PT X,Y+1
63C1:	60			424		RTS			
				425	*				
				426	*PRINT	SUBROUTINE			
				427	*				
63C2:	A2	00		428	PRINT	LDX	#\$00		
63C4:	A9	05		429		LDA	#\$05		
63C6:	85	24		430		STA	\$24		;HTAB5
63C8:	A9	17		431		LDA	#\$17		

```

63CA: 20 5B FB 432      JSR  TABV      ;VTAB23
63CD: BD E0 63 433 PRINT1 LDA  STRING,X
63D0: F0 07   434      BEQ  DONE
63D2: 20 F0 FD 435      JSR  COUT
63D5: E8     436      INX
63D6: 4C CD 63 437      JMP  PRINT1
63D9: AD 10 60 438 DONE  LDA  SUM
63DC: 20 DA FD 439      JSR  PRBYTE
63DF: 60     440      RTS
63E0: D3 C3 CF
63E3: D2 C5 A0
63E6: BD A0   441 STRING  ASC  "SCORE = "
63E8: 00     442      HEX  00
        443 *
        444 *SOUND SUBROUTINE
        445 *
63E9: AD 30 C0 446 SOUND  LDA  $C030
63EC: 88     447 S1      DEY
63ED: D0 05   448      BNE  S2
63EF: CE 16 60 449      DEC  TIME
63F2: F0 09   450      BEQ  SEND
63F4: CA     451 S2      DEX
63F5: D0 F5   452      BNE  S1
63F7: AE 15 60 453      LDX  PITCH
63FA: 4C E9 63 454      JMP  SOUND
63FD: 60     455 SEND    RTS

```

```
--END ASSEMBLY--      1022 BYTES
```

MACHINE LANGUAGE ACCESS TO APPLESOFT HI-RES ROUTINES

The Applesoft ROM contains a full set of Hi-Res graphics routines. But Applesoft, being an interpretive language rather than a compiled language, accesses these routines rather inefficiently as far as speed is concerned. This is because the interpreter has to determine where to go and what to do with each tokenized BASIC instruction as it encounters it. The speed penalty for this added overhead is considerable. The interpreter runs these routines from four to six times slower than if they were called directly from machine language.

At first glance, it appears to be rather simple to call to graphics subroutines located in the ROM. In retrospect, it is, provided that you understand how the interpreter handles the data structure both internally and externally as it executes these graphics subroutines. Since the information has never been fully documented, it is some help if you have the Programmer's Aid Manual, where a source listing of that ROM chip is quite similar to the ROM Applesoft Hi-Res subroutines.

I'm quite reluctant at this stage to attempt an explanation of how these routines actually work. A solid grounding both in machine language and in the Hi-res screen's peculiarities won't come until much later in the book. I will, however, discuss the data structure in regards to what you need to input, and how you input these parameters when calling the subroutines.

There are a series of memory locations stored in zero page that specify a point on the Hi-Res screen. Some people call these locations External Cursor Data. They are as follows:

- \$E0:** Lo order byte of the horizontal screen coordinate
- \$E1:** Hi order byte of the horizontal screen coordinate
- \$E2:** Vertical screen coordinate
- \$E4:** Color masking word from the color table (\$F6F6-\$F6FD)
- \$E6:** Page indicator (\$20 page 1, \$40 for page 2).

In addition, three other memory locations hold information regarding shape table data for the drawing subroutines:

- \$E7:** Scale factor for drawing shapes
- \$E8:** Lo byte pointer to beginning of shape table
- \$E9:** Hi byte pointer to beginning of shape table.

There are also a number of zero page locations that the Hi-Res subroutines use internally when doing the actual screen plotting of points, or strings of points called lines. Some of these contain the memory address of the byte to plot on the screen, while others contain the color and masking information, so that only the correct pixel within that seven-pixel byte is turned on or off.

\$1C: The color masking byte, which is shifted for odd addresses but otherwise remains unchanged.

\$26: Lo address for the leftmost byte in a particular vertical row.

\$27: Hi address for the leftmost byte in a particular vertical row.

\$E5: The integer part of the horizontal screen coordinate divided by 7, or the horizontal offset into row.

\$30: The bit position taken from the Bit Position table.

This corresponds to remainder from horizontal coordinate divided by 7 or which bit in the byte is to be lit.

What I should point out is that after a series of other subroutines set up the position to plot on the screen, the actual plotting of the point is done with a five line subroutine called PLOT located at **\$F45A**, as in the following:

```
LDA    $1C
EOR    ($26),Y
AND    $30
EOR    ($26),Y
STA    ($26),Y
RTS
```

The internal cursor data is more important than the external cursor data if speed is the consideration. There are internal subroutines within the ROM that set the external cursor data to correspond with the internal data, and several more that can manipulate the screen cursor directly. However, for plotting points and drawing shapes from Apple shape tables, you need not concern yourself with any internal workings of these subroutines. Instead, I've summarized all of the necessary subroutines in the table below, and will demonstrate examples using them.

NAME	ADDRESS	ACC.	XREG	YREG	NOTES
HGR	\$F3E2	-----	----		
HGR2	\$F3D8	----	----	----	
BKGND	\$F3F4	COLOR FROM COLOR MASK TABLE	----	----	
HCOLOR	\$F6F0	----	COLOR 0-7	----	
HPLOT	\$F457	VERT	HORIZ LO	HORIZ HI	THIS CALLS HPOSN
HLINE	\$F53A	HORIZ LO	HORIZ HI	VERT	DRAWS FROM INT CURSOR POS. TO PT. IN INPUT
HPOSN	\$F411	VERT	HORIZ LO	HORIZ HI	ALWAYS CALL BEFORE DRAW
SHPTR	\$F730	----	SHAPE #	----	SETS \$1A, \$1B SHAPE POINTERS
DRAW	\$F601	ROTATION	\$1A	\$1B	
XDRAW	\$F65D	ROTATION	\$1A	\$1B	

Simple shapes can be plotted to the Hi-Res screen in BASIC by HPLOTting from point to point. Their speed, in comparison to Apple shapes (vector shapes), is rather slow. However, in machine code, HPLOTed shapes become a viable alternative if the shape is rather large and complex. Their disadvantage is that they can't be scaled or rotated, but they are easier to plot if you choose to place the coordinate pairs into a table.

Our first example will plot a simple triangle by accessing the Applesoft Hi-Res ROM routines directly. It is equivalent to the following BASIC program.

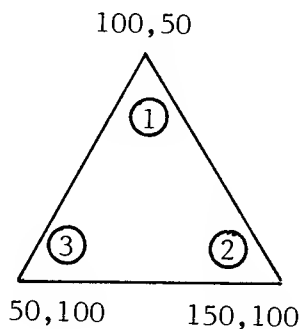

```

10 HGR
20 HCOLOR = 3
30 HPLOT 100,50 TO 150,100 TO 50,100 TO 100,50
40 END

```

The program sets the mode to Hi-Res graphics page one, mixed text and graphics, by calling HGR at \$F3E2. The plotting color is set to white (3) by a call to HCOLOR at \$F6F0. Then, by loading the Accumulator and the X & Y registers with the correct screen coordinates, the point at 100,50 is plotted to the screen with a call to HPLOT at \$F457. Each of the triangle's lines are drawn by calling HLINE at \$F53A. This subroutine draws a line from the internal cursor position (last point) to the point defined by the input to HLINE. Since the last point was at 100,50 and we are inputting the coordinates 150,100, the line is drawn between these two points. After drawing the next two lines, the triangle is completed and the program ends. The complete code follows.

IMPORTANT NOTE: The programs in this chapter access the Applesoft ROM. While this is no problem to Apple II Plus owners, those of us that have an Integer machine with an Applesoft ROM card, or Applesoft in RAM on a 16K memory board, should understand that if they enter the monitor by hitting reset, they have lost Applesoft. The machine reverts to the Integer ROM on the motherboard. If you try to restart the programs they won't run unless the ROMs are reconnected by a 9DBFG and you return to the monitor by a CALL -151.



```

1      *PLOT TRIANGLE
2      ORG  $6000
6000: 20 E2 F3 3      JSR  $F3E2      ;HGR
6003: A2 03 4      LDX  #$03      ;COLOR=WHITE
6005: 20 F0 F6 5      JSR  $F6F0      ;HCOLOR
6      *PLOT FIRST PT
6008: A0 00 7      LDY  #$00      ;HORIZ POS HI BYTE
600A: A2 64 8      LDX  #$64      ;HORIZ POS LO BYTE
600C: A9 32 9      LDA  #$32      ;VERT POS
600E: 20 57 F4 10     JSR  $F457      ;HPLLOT
11     *DRAW TO SECOND POINT
6011: A2 00 12     LDX  #$00      ;HORIZ POS HI BYTE
6013: A9 96 13     LDA  #$96      ;HORIZ POS LO BYTE
6015: A0 64 14     LDY  #$64      ;VERT POS
6017: 20 3A F5 15     JSR  $F53A      ;HLINE
16     *DRAW TO THIR D POINT
601A: A2 00 17     LDX  #$00      ;HORIZ POS HI BYTE
601C: A9 32 18     LDA  #$32      ;HORIZ POS LO BYTE
601E: A0 64 19     LDY  #$64      ;VERT POS
6020: 20 3A F5 20     JSR  $F53A      ;HLINE
21     *DRAW TO FIRST POINT
6023: A2 00 22     LDX  #$00      ;HORIZ POS HI BYTE
6025: A9 64 23     LDA  #$64      ;HORIZ POS LO BYTE
6027: A0 32 24     LDY  #$32      ;VERT POS
6029: 20 3A F5 25     JSR  $F53A      ;HLINE
602C: 60 26      RTS

```

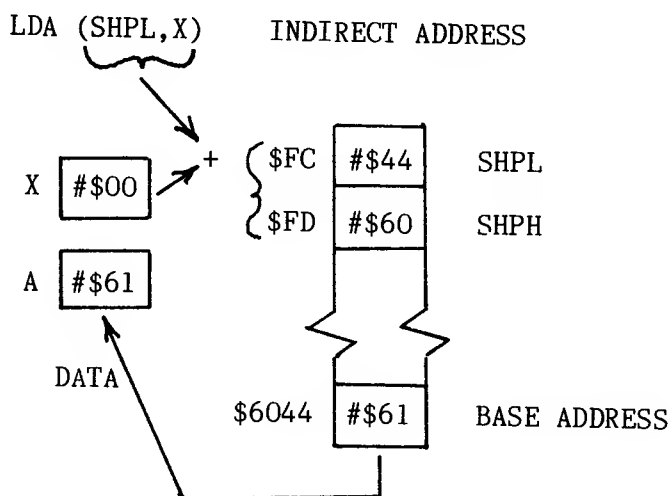
--END ASSEMBLY--

The HPLLOT technique can be used to draw shapes of greater complexity. Since these shapes require numerous calls to HLINE for each line segment of the completed shape, it is best to design the code to access the coordinate pairs from a stored table and put the drawing routine into a loop.

For the sake of simplicity, I decided to store the X-Y coordinates as two byte pairs. This limits the range along the horizontal axis, since values greater than 255 would require using the hi byte, too. If you wanted to use the entire screen, you would have to use three byte coordinate pairs and modify the code accordingly. A test was needed to determine when all the shape's points had been plotted. I used an \$FF as a flag for the last point. The test is on the vertical coordinate, since Y coordinate values don't exceed \$BF. Actually, the pair's first byte can be anything, since it is the last byte of the pair that is the flag. When the loop detects this flag, it skips plotting the last line segment and exits the loop.

The technique for accessing elements of a shape table involves loading the first of a pair of bytes into the Accumulator, and the second byte into the X register before calling HLINE to draw the line segment. Each element of the table is stored at a particular two-byte address. In our example, the very first element is called the 0th element of the table and is located at \$6044. Elements of a table can be accessed by using a zero page indexing system called Indexed Indirect Addressing. It takes the form LDA (SHPL,X). If the X-register were zero, it would load a byte from an address indicated by a pair of bytes, SHPL and SHPH stored in zero page. For example, if location \$FC and \$FD, which are equivalent to SHPL and SHPH respectively, contain a #\$44 and #\$60 in that order, then LDA (SHPL,X) will load a #\$61 from location \$6044 into the Accumulator.

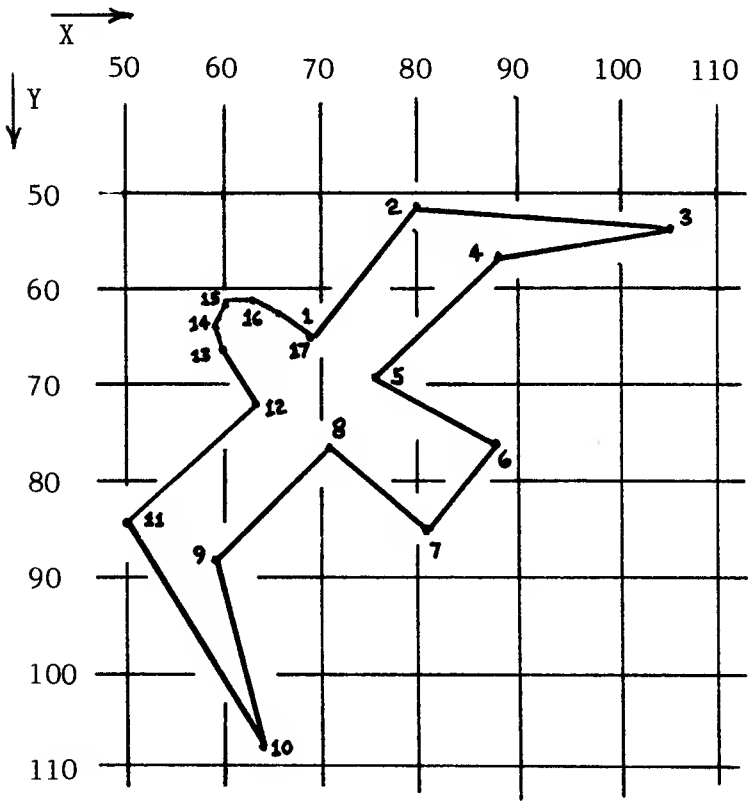
INDEXED INDIRECT ADDRESSING



As you will soon discover, there are never enough registers in the 6502. Certainly, the Accumulator and X and Y registers are not enough when all three need to be loaded to call a subroutine, and you also need to use two of them simultaneously for retrieving data from a table. The solution is to temporarily store your data in a memory location. When you're done with the table and your registers are free, the data can be moved to the proper registers just before calling the subroutine. The important thing is to be careful that you do not clobber your working registers.

In the example below, the X-register must be set to zero each time the indexed indirect load is used to retrieve a value from the table. This is no problem the first time through the loop, but this value for the horizontal position lo byte eventually needs to reside in the X-register before calling HLINE. Since we

need to do another indirect indexed load using both the Accumulator and X-register for the next byte, we temporarily store our data in XLOW. If we increment SHPL, the lo byte pointer to our shape data, it will point to the next byte in our shape table. At this point, since we haven't disturbed the X-register, we don't need to put zero into it to perform our next indirect indexed load. This second value retrieved — the vertical coordinate is transferred to the Y-register. The horizontal hi byte is placed into the X-register and the horizontal lo byte, which was temporarily stored at XLOW, is moved into the Accumulator before calling the subroutine HLINE.



DECIMAL			HEX	
PT	X	Y	X	Y
1	69	65	45	41
2	80	52	50	34
3	106	57	6A	39
4	87	57	57	39
5	76	71	4C	47
6	88	77	58	4D
7	81	85	51	55
8	72	77	48	40
9	59	88	38	58
10	64	108	40	6C
11	50	84	32	54
12	63	72	3F	48
13	59	67	3B	43
14	58	64	3A	40
15	60	62	3C	3E
16	64	62	40	3E
17	69	65	44	41
			FF	FF

```

1  *HPLOTS A BIRD SHAPE ON SCREEN ONCE
2
3  XLOW DS 1
4  HPLOT EQU $F457
5  HLINE EQU $F53A
6  HCOLOR EQU $F6F0
7  HGR EQU $F3E2
8  SHPL EQU $FC
9  SHPH EQU SHPL+$1
10 *PROGRAM
6001: 20 E2 F3 11 JSR HGR
6004: A2 03 12 LDX #$03 ;WHITE COLOR
6006: 20 F0 F6 13 JSR HCOLOR ;SET WHITE COLOR
6009: A9 44 14 LDA #<SHAPE
600B: 85 FC 15 STA SHPL
600D: A9 60 16 LDA #>SHAPE
600F: 85 FD 17 STA SHPH
18 *PLOT FIRST POINT
6011: A2 00 19 PLOT LDX #$00
6013: A1 FC 20 LDA (SHPL,X) ;THIS IS HOR POS LO BYTE
6015: 8D 00 60 21 STA XLOW
6018: E6 FC 22 INC SHPL ;NEXT BYTE IN SHAPE TABLE
601A: A1 FC 23 LDA (SHPL,X) ;THIS IS VERT VALUE FOR PT
601C: AE 00 60 24 LDX XLOW ;HORIZ POS LO BYTE
601F: A0 00 25 LDY #$00 ;HORIZ POS HI BYTE
6021: 20 57 F4 26 JSR HPLOT
6024: E6 FC 27 INC SHPL ;NEXT BYTE IN TABLE
28 *DRAW NEXT POINT

```

```

6026: A2 00    29  LOOP      LDX  #$00
6028: A1 FC    30          LDA  (SHPL,X)  ;HORIZ POS LO BYTE
602A: 8D 00 60 31          STA  XLOW
602D: E6 FC    32          INC  SHPL      ;NEXT BYTE IN TABLE
602F: A1 FC    33          LDA  (SHPL,X)  ;THIS IS VERT VALUE FOR PT
6031: C9 FF    34          CMP  #$FF
6033: F0 OE    35          BEQ  DONE      ;IF BYTE CONTAINS 255, DONE
6035: A8       36          TAY          ;VERT IN Y REG
6036: A2 00    37          LDX  #$00      ;HORIZ POS IN HI BYTE
6038: AD 00 60 38          LDA  XLOW      ;HORIZ POS IN LO BYTE
603B: 20 3A F5 39          JSR  HLINE
603E: E6 FC    40          INC  SHPL      ;NEXT BYTE
6040: 4C 26 60 41          JMP  LOOP
6043: 60       42          DONE  RTS
        43          *
6044: 45 41 50
6047: 34 6A 39
604A: 57 39    44  SHAPE    HEX  454150346A395739
604C: 4C 47 58
604F: 4D 51 55
6052: 48 4D    45          HEX  4C47584D5155484D
6054: 3B 58 40
6057: 6C 32 54
605A: 3F 48    46          HEX  3B58406C32543F48
605C: 3B 43 3A
605F: 40 3C 3E
6062: 40 3E    47          HEX  3B433A403C3E403E
6064: 44 41 FF
6067: FF       48          HEX  4441FFFF

```

Shape tables that cross page boundaries (256 byte sections of memory where the hi byte is constant) can cause problems. If, for example, our table began at \$60FC instead of \$6044, after incrementing four times, the lo byte would be \$00. The program would attempt to load the byte at location \$6000 instead of the byte at location \$6100. This can be prevented if a test is performed after you increment SHPL. If SHPL were equal to zero, it would increment SHPH; otherwise, it would skip this step.

```

        INC  SHPL      ;INCREMENT LO BYTE
        LDA  SHPL
        CMP  #$00      ;IS IT 0 ?
        BNE  SKIP      ;NO
        INC  SHPH      ;YES INCREMENT HI POINTER
SKIP    LDA  (SHPL,X)  ;NEXT BYTE IN TABLE
        :
        :

```

The object of this fast machine language algorithm is to enable you to animate your shapes smoothly and quickly. While one would never attempt to animate HPLoTted shapes in Applesoft BASIC, it is completely feasible in machine language. Speed increases on the order of 6 to 8 times are the rule.

The code to animate our HPLOTed bird in Applesoft follows. Try it, then try the same algorithm written in machine language. I should point out that the speed differences can not be directly correlated, since to keep the object on the screen longer than off, a delay loop of 7 milliseconds per frame was used. If you remove the delay or set the value in the Accumulator to #\$01 before calling the delay subroutine at \$FCA8, the speed increases to 8 times that of the Applesoft version. However, screen flicker becomes more noticeable.

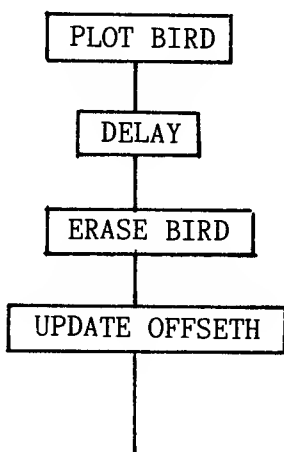
```

10 DIM X(20),Y(20)
30 FOR I = 1 TO 50
40 READ X(I),Y(I)
50 IF Y(I) = 255 THEN 65
60 NEXT I
65 HGR :OFF = - 50:I = 1
70 HCOLOR= 3
80 HPLOT X(I) + OFF,Y(I) TO X(I + 1) + OFF,Y(I + 1) TO X(I
+ 2) + OFF,Y(I + 2) TO X(I + 3) + OFF,Y(I + 3) TO X(I + 4) +
OFF,Y(I + 4) TO X(I + 5) + OFF,Y(I + 5) TO X(I + 6) + OFF,Y
(I + 6) TO X(I + 7) + OFF,Y(I + 7) TO X(I + 8) + OFF,Y(I + 8
) TO X(I + 9) + OFF,Y(I + 9)
90 HPLOT X(I + 9) + OFF,Y(I + 9) TO X(I + 10) + OFF,Y(I + 1
0) TO X(I + 11) + OFF,Y(I + 11) TO X(I + 12) + OFF,Y(I + 12)
TO X(I + 13) + OFF,Y(I + 13) TO X(I + 14) + OFF,Y(I + 14) T
O X(I + 15) + OFF,Y(I + 15) TO X(I + 16) + OFF,Y(I + 16)
100 HCOLOR= 4
110 HPLOT X(I) + OFF,Y(I) TO X(I + 1) + OFF,Y(I + 1) TO X(I
+ 2) + OFF,Y(I + 2) TO X(I + 3) + OFF,Y(I + 3) TO X(I + 4)
+ OFF,Y(I + 4) TO X(I + 5) + OFF,Y(I + 5) TO X(I + 6) + OFF,
Y(I + 6) TO X(I + 7) + OFF,Y(I + 7) TO X(I + 8) + OFF,Y(I +
8) TO X(I + 9) + OFF,Y(I + 9)
120 HPLOT X(I + 9) + OFF,Y(I + 9) TO X(I + 10) + OFF,Y(I +
10) TO X(I + 11) + OFF,Y(I + 11) TO X(I + 12) + OFF,Y(I + 12
) TO X(I + 13) + OFF,Y(I + 13) TO X(I + 14) + OFF,Y(I + 14)
TO X(I + 15) + OFF,Y(I + 15) TO X(I + 16) + OFF,Y(I + 16)
130 OFF = OFF + 5
140 IF OFF = 155 THEN OFF = - 50
150 GOTO 70
160 DATA 69,65,80,52,106,57,87,57,76,71,88,77,81,85,72,77
,59,88,64,108,50,84,63,72,59,67,58,64,60,62,64,62,69,65,255,
255

```

The code for the moving bird is quite similar to the stationary bird, except that once we plot the bird, it must be erased before replotting it at a different position. It becomes rather convenient to place the entire plotting program in a subroutine. An offset is added to each horizontal point of the bird to position it properly on the screen. This offset starts at -50 or `#$CE` in order to position the bird's left-most point at $X = 0$. The offset is incremented by five for each additional frame and tested each time so that it doesn't exceed 150 or `#$96`. If it does, the bird's right-most point will exceed 255 decimal. The test must be exactly at 150 rather than equal or greater, because our negative numbers `#$CE` and larger would also meet the test. Be careful in this kind of test. If your hexadecimal addition isn't correct when choosing the test position, the number will never meet the test conditions and therefore never reset the offset back to the beginning position after traversing the screen's width. One hint is to use the monitor when adding two hexadecimal single byte numbers. For example, the monitor command `03 + FE <CR>` will return the hexadecimal value `$02`.

When alternating between drawing and erasing, the color shifts between white and black, respectively. The pointers to the shape table must also be reset for each plot/erase cycle because these pointers are incremented when retrieving bytes within the table. The flow chart and machine code for the moving bird follows.




```

1      *MOVING HPLOTTED BIRD ACROSS SCREEN
2      ORG    $6000
3      XLOW   DS    1
4      HPLLOT EQU    $F457
5      HLINE  EQU    $F53A
6      HCOLOR EQU    $F6F0
7      HGR    EQU    $F3E2
8      SHPL   EQU    $FC
9      SHPH   EQU    SHPL+$1
10     OFFSETH DS    1
11     *PROGRAM
6002: 20 E2 F3 12      JSR    HGR
6005: A9 CE      13      LDA    #$CE          ; -50 DECIMAL
6007: 8D 01 60 14      STA    OFFSETH
600A: A9 7C      15      MAIN    LDA    #<SHAPE
600C: 85 FC      16      STA    SHPL
600E: A9 60      17      LDA    #>SHAPE
6010: 85 FD      18      STA    SHPH
6012: A2 03      19      LDX    #$03          ; WHITE COLOR
6014: 20 F0 F6 20      JSR    HCOLOR          ; SET TO WHITE
6017: 20 41 60 21      JSR    PLOT
601A: A9 50      22      LDA    #$50
601C: 20 A8 FC 23      JSR    $FCA8          ; DELAY
601F: A9 7C      24      LDA    #<SHAPE
6021: 85 FC      25      STA    SHPL
6023: A9 60      26      LDA    #>SHAPE
6025: 85 FD      27      STA    SHPH
6027: A2 04      28      LDX    #$04          ; BLACK COLOR
6029: 20 F0 F6 29      JSR    HCOLOR          ; SET TO BLACK
602C: 20 41 60 30      JSR    PLOT
31     *UPDATE HORIZ OFFSET
602F: AD 01 60 32      LDA    OFFSETH
6032: 18        33      CLC
6033: 69 05      34      ADC    #$05
6035: C9 96      35      CMP    #$96          ; 150 DECIMAL
6037: D0 02      36      BNE    SKIP
6039: A9 CE      37      LDA    #$CE          ; OFF RT SIDE OF SCREEN
603B: 8D 01 60 38      SKIP    STA    OFFSETH
603E: 4C 0A 60 39      JMP    MAIN
40     *PLOT FIRST POINT
6041: A2 00      41      PLOT    LDX    #$00
6043: A1 FC      42      LDA    (SHPL,X)          ; THIS IS HOR POS LO BYTE
6045: 18        43      CLC
6046: 6D 01 60 44      ADC    OFFSETH
6049: 8D 00 60 45      STA    XLOW          ; NEW HORIZ POS LO BYTE
604C: E6 FC      46      INC    SHPL          ; NEXT BYTE IN SHAPE TABLE
604E: A1 FC      47      LDA    (SHPL,X)          ; THIS IS VERT VALUE FOR PT
6050: AE 00 60 48      LDX    XLOW          ; HORIZ POS LO BYTE
6053: A0 00      49      LDY    #$00          ; HORIZ POS HI BYTE
6055: 20 57 F4 50      JSR    HPLLOT
6058: E6 FC      51      INC    SHPL          ; NEXT BYTE IN TABLE
52     *DRAW NEXT POINT
605A: A2 00      53      LOOP    LDX    #$00
605C: A1 FC      54      LDA    (SHPL,X)          ; HORIZ POS LO BYTE
605E: 18        55      CLC
605F: 6D 01 60 56      ADC    OFFSETH
6062: 8D 00 60 57      STA    XLOW          ; NEW HORIZ POS LO BYTE
6065: E6 FC      58      INC    SHPL          ; NEXT BYTE IN TABLE
6067: A1 FC      59      LDA    (SHPL,X)          ; THIS IS VERT VALUE FOR PT
6069: C9 FF      60      CMP    #$FF

```

606B:	F0	OE	61		BEQ	DONE		;IF BYTE CONTAINS 255, DONE
606D:	A8		62		TAY			;VERT IN Y REG
606E:	A2	00	63		LDX	#\$00		;HORIZ POS IN HI BYTE
6070:	AD	00	60	64	LDA	XLOW		;HORIZ POS IN LO BYTE
6073:	20	3A	F5	65	JSR	HLINE		
6076:	E6	FC	66		INC	SHPL		;NEXT BYTE
6078:	4C	5A	60	67	JMP	LOOP		
607B:	60		68		RTS			
			69	*				
607C:	45	41	50					
607F:	34	6A	39					
6082:	57	39	70	SHAPE	HEX	454150346A395739		
6084:	4C	47	58					
6087:	4D	51	55					
608A:	48	4D	71		HEX	4C47584D5155484D		
608C:	3B	58	40					
608F:	6C	32	54					
6092:	3F	48	72		HEX	3B58406C32543F48		
6094:	3B	43	3A					
6097:	40	3C	3E					
609A:	40	3E	73		HEX	3B433A403C3E403E		
609C:	44	41	FF					
609F:	FF		74		HEX	4441FFFF		

--END ASSEMBLY-- 160 BYTES

APPLE SHAPE TABLES IN ANIMATION

The advantage of accessing Apple shape tables (vector shape tables) directly from machine language results in a sixfold increase in animation speed. For many applications and simple games, this speed increase may be sufficient. If it isn't, you should use raster or block shape animation.

I think that beginning machine language programmers, whose prior experience is with Apple shapes in BASIC, should attempt the techniques in this section before learning more complicated methods shown later in this book.

If you were to DRAW or XDRAW a shape in BASIC, you would set the color, scale, and rotation before doing a DRAW 1 at 10,10. The location of the shape table would have been indicated by poking the address to locations decimal 232 and 233. These two locations are \$E8 and \$E9, respectively.

However, before calling the DRAW subroutine at \$F601 or XDRAW at \$F65D, the pointers to the correct shape number must be set through a subroutine that I call SHPTR (short for shape pointer). This subroutine located at \$F730 takes the shape number, which is inputted via the X-register, and sets the pointers to the shape in locations \$1A (lo byte) and \$1B (hi byte).

This subroutine is deeply linked into the Applesoft interpreter. It calls subroutines that increment the Applesoft "Get Next Character" Routine. Although I don't believe that this subroutine located at \$B7 will cause any pro-

blems, before you clobber anything, I would pay attention to the chart of available zero page locations in the Apple Reference Manual. Don't touch the locations used by Applesoft. You can also disconnect that routine by placing a #\$60 (RTS) in location \$B7 (its first location), but be sure to put the original value, #\$AD, back when you're done, or you will hang the computer when it returns the Applesoft prompt, and doesn't understand anything that you type. In short, don't make the change unless you think it is causing you grief.

The second thing that must be set before calling the DRAW subroutine is the internal cursor position, or where you want to plot your shape. This is easily accomplished with the HPOSN subroutine at \$F411. Once the horizontal and vertical locations are inputted, the subroutine sets locations \$26, \$27, \$30, and \$E5 to begin plotting. When you finally call the DRAW or XDRAW subroutine, the only inputs that are required are the rotation value in the Accumulator and the pointers to the correct shape that are stored at \$1A and \$1B in the X and Y registers. It may sound complicated but if you examine the following code, you will see that it is relatively straight-forward. The following routine XDRAWs two shapes. The first, a square, is plotted at X = 64, Y = 64, and the second shape, a cross, is plotted at X = 128, Y = 50. The scale is 4.

```

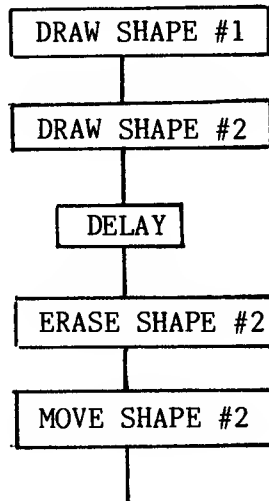
1      *PLOTS TWO APPLE SHAPE TABLE SHAPES
2      ORG    $6000
3      HGR     EQU    $F3E2
4      HCOLOR  EQU    $F6F0
5      HPOSN   EQU    $F411
6      XDRAW   EQU    $F65D
7      SHPTR   EQU    $F730
6000: 20 E2 F3 8      JSR    HGR
6003: A9 00 9        LDA    #$00
6005: 85 E8 10       STA    $E8           ;LO BYTE OF SHAPE TABLE
6007: A9 08 11       LDA    #$08
6009: 85 E9 12       STA    $E9           ;HI BYTE OF SHAPE TABLE
600B: A2 03 13       LDX    #$03           ;WHITE
600D: 20 F0 F6 14    JSR    HCOLOR
6010: A9 02 15       LDA    #$02
6012: 85 E7 16       STA    $E7           ;SCALE
6014: A2 01 17       LDX    #$01           ;SHAPE #1
6016: 20 30 F7 18    JSR    SHPTR         ;SET UP POINTER TO 1ST SHAPE
6019: A2 40 19       LDX    #$40           ;HOR LO
601B: A0 00 20       LDY    #$00           ;HOR HI
601D: A9 40 21       LDA    #$40           ;VERT
601F: 20 11 F4 22    JSR    HPOSN
6022: A6 1A 23       LDX    $1A           ;LO BYTE SHAPE ADDRESS
6024: A4 1B 24       LDY    $1B           ;HI BYTE SHAPE ADDRESS
6026: A9 00 25       LDA    #$00           ;ROT
6028: 20 5D F6 26    JSR    XDRAW
27      *PLOT SECOND SHAPE
602B: A2 02 28       LDX    #$02           ;SHAPE #2
602D: 20 30 F7 29    JSR    SHPTR         ;SET UP POINTER TO 2ND SHAPE
6030: A2 80 30       LDX    #$80           ;HOR LO
6032: A0 00 31       LDY    #$00           ;HOR HI
6034: A9 32 32       LDA    #$32           ;VERT

```

6036:	20 11 F4 33	JSR	HPOSN	
6039:	A6 1A 34	LDX	\$1A	;LO BYTE SHAPE ADDRESS
603B:	A4 1B 35	LDY	\$1B	;HI BYTE SHAPE ADDRESS
603D:	A9 00 36	LDA	#\$00	;ROT
603F:	20 5D F6 37	JSR	XDRAW	
6042:	60 38	RTS		

--END ASSEMBLY-- 67 BYTES

Animating a shape is simple. You plot it once, erase it, move it to a new position, and then replot it at its new position. The procedure is accomplished via a loop. There is very little to say about the method. It is the same in Applesoft. I think the only thing you should be aware of is that HPOSN doesn't need to be called twice, since the erase is done at the same screen position as the XDRAW. In the example, shape #2 moves horizontally to the right, while shape #1 is stationary. The move routine checks for wrap-a-round at X = #\$FF as it moves the shape across the screen. The flow chart and code follows.



SHAPE #1



SHAPE #2



SHAPE @ \$800

SHAPE TABLE: 02 00 06 00 09 00 2C 3E 00 2C 2E 3E 3E 3C 2C 00

TWO SHAPES
 OFFSET TO SHAPE #1
 OFFSET TO SHAPE #2
 SHAPE #1
 SHAPE #2

```

1  *MOVES APPLE SHAPE TABLE SHAPE ACROSS SCREEN
2      ORG $6000
3  HGR      EQU $F3E2
4  HCOLOR   EQU $F6F0
5  HPOSN    EQU $F411
6  XDRAW    EQU $F65D
7  SHPTR    EQU $F730
8  XLOW     DS 1
6001: A9 05 9      LDA #$05
6003: 8D 00 60 10   STA XLOW
6006: 20 E2 F3 11   JSR HGR
6009: A9 00 12      LDA #$00
600B: 85 E8 13      STA $E8      ;LO BYTE OF SHAPE TABLE
600D: A9 08 14      LDA #$08
600F: 85 E9 15      STA $E9      ;HI BYTE OF SHAPE TABLE
6011: A2 03 16      LDX #$03      ;WHITE
6013: 20 F0 F6 17   JSR HCOLOR
6016: A9 04 18      LDA #$04
6018: 85 E7 19      STA $E7      ;SCALE
601A: A2 01 20      LDX #$01      ;SHAPE #1
601C: 20 30 F7 21   JSR SHPTR    ;SET UP POINTER TO 1ST SHAPE
601F: A2 40 22      LDX #$40      ;HORIZ POS LO BYTE
6021: A0 00 23      LDY #$00      ;HORIZ POS HI BYTE
6023: A9 50 24      LDA #$50      ;VERT POS
6025: 20 11 F4 25   JSR HPOSN
6028: A6 1A 26      LDX $1A      ;LO BYTE SHAPE ADDRESS
602A: A4 1B 27      LDY $1B      ;HI BYTE SHAPE ADDRESS
602C: A9 00 28      LDA #$00
602E: 20 5D F6 29   JSR XDRAW    ;ROT
30  *PLOT SECOND SHAPE
6031: A2 02 31      LOOP LDX #$02 ;SHAPE #2
6033: 20 30 F7 32   JSR SHPTR    ;SET UP POINTER TO 2ND SHAPE
6036: AE 00 60 33   LDX XLOW      ;HOR POS LO BYTE
6039: A0 00 34      LDY #$00      ;HOR POS HI BYTE
603B: A9 32 35      LDA #$32      ;VERT POS
603D: 20 11 F4 36   JSR HPOSN
6040: A6 1A 37      LDX $1A      ;LO BYTE SHAPE ADDRESS
6042: A4 1B 38      LDY $1B      ;HI BYTE SHAPE ADDRESS
6044: A9 00 39      LDA #$00      ;ROT
6046: 20 5D F6 40   JSR XDRAW    ;DRAW SHAPE #2
6049: A9 50 41      LDA #$50
604B: 20 A8 FC 42   JSR $FCA8    ;DELAY
604E: A2 02 43      LDX #$02      ;SHAPE #2
6050: 20 30 F7 44   JSR SHPTR
45  *DON'T HAVE TO DO HPOSN BEFORE ERASE
46  *BECAUSE POSITION HASN'T CHANGED
6053: A6 1A 47      LDX $1A      ;LO BYTE SHAPE ADDRESS
6055: A4 1B 48      LDY $1B      ;HI BYTE SHAPE ADDRESS

```

6057:	A9	00	49		LDA	#\$00		;ROT
6059:	20	5D	F6	50		JSR	XDRAW	;ERASE SHAPE #2
			51		*MOVE	SHAPE	TO	NEW POSITION
605C:	AD	00	60	52		LDA	XLOW	
605F:	18			53		CLC		
6060:	69	05		54		ADC	#\$05	
6062:	C9	FF		55		CMP	#\$FF	
6064:	D0	02		56		BNE	SKIP	
6066:	A9	0A		57		LDA	#\$0A	
6068:	8D	00	60	58	SKIP	STA	XLOW	
606B:	4C	31	60	59		JMP	LOOP	

HI-RES SCREEN ARCHITECTURE

The Apple II has two Hi-Res graphics screens, a primary and a secondary, each with a resolution of 280 dots horizontally (columns) and 192 dots or lines vertically. This gives an effective screen resolution of 53,760 picture elements or pixels per screen.

The large number of pixels presented a dilemma to the Apple II designers. Using one memory location for each dot would far outstrip the Apple's 48K memory; besides, they wanted to have two screens. Their solution was to divide the screen horizontally into 40 groups of 7 pixels. Each memory location would represent information for seven adjacent pixels. This lowered the memory requirement to 7680 bytes per screen. Since it was easier to work in 8K blocks of memory, this left an unused 512 bytes of memory per page.

In 1977, when memory chips were expensive, most Apple II computers were sold with only 16K of memory. With various monitor areas, zero page, the stack, and the text page using the first 2K (2048) bytes of memory, it seemed logical to place Hi-Res graphics screen # one at the upper end of memory, locations 8192 to 16383 (\$2000- \$3FFF). Screen # two of Hi-Res graphics was placed in the 8K block of memory just beyond locations 16384 to 24575 (\$4000-\$5FFF). It was usable by owners who purchased extra memory. Both of these screen's locations are hardwired into the machine and, unfortunately, are not relocatable. In those days, before DOS and Applesoft made their debut, Integer BASIC programmers whose machines contained 48K of memory could start their program at the top of memory and write 32K of code.

Today, Applesoft programmers face the dilemma of where to place their programs without overwriting the information stored in the Hi-Res screen areas. Since Applesoft loads a program immediately above the text screen which begins at \$800 or 2048 decimal, only small programs fit, if they are using Hi-Res graphics commands. The solution is to set the Applesoft pointers so that the program loads above the Hi-Res screen. Unfortunately, you waste the 6K of usable memory between the operating system and the beginning of Hi-Res screen one. In retrospect, what seemed to be a logical choice in 1977 is cumbersome today.

The Apple's Hi-Res screen is considered memory-mapped. If you were to change the values of the first 40 bytes of screen memory so that each turned on all 7 pixels, then the screen would display a solid white line at the top. Changing any particular byte in Hi-Res memory directly affects the resultant picture.

Any byte in screen memory consists of a sequence of eight individual bits. If a bit is on, it has a value of 1; if it is off, it has a value of 0. This on-off system of numbers is called "Binary". Binary numbers, represented by strings of 0's and 1's, have their least significant numbers starting at the right, as shown:

128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	1	= \$01

Each successive move of a bit to the left results in the value of the byte being multiplied by two.

128	64	32	16	8	4	2	1	
0	0	0	0	0	0	1	0	= \$02

Eventually, the on bit would be shifted to the far left with a value of \$80 or 128 decimal.

The Hi-Res screen's convention is in reverse. Pixel values increase from left to right. This can be verified by poking values into the primary screen's first memory location, \$2000. To do this it, is best to enter the monitor with a CALL -151 from BASIC. Hi-Res graphics with mixed text can be invoked with the following commands:

*C050	<CR>	SET GRAPHICS MODE
*C053	<CR>	SET MIXED TEXT AND GRAPHICS
*C057	<CR>	SET HI-RES GRAPHICS









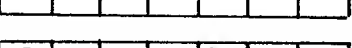

Most likely, the screen is not clear. Although an HGR from Applesoft would clear it before entering the monitor, you should learn to perform this operation from the monitor. Typing a 2000:00 <CR> will place a zero or no lit pixels in the first screen location. Doing the following memory move shifts the 0 to all other locations in a cascade effect on Hi-Res screen page one:

***2001<2000.3FFFFM <CR>**

If you enter 2000:01 <CR>, a single dot appears at the top left. If you enter 2000:02 <CR>, the dot moves one position to the right. A 2000:04 <CR> moves it right once again. Since seven dots are controlled by one byte, you can do this seven times. The value \$40 shifts it to the seventh position. If you shift the dot one extra time with the value \$80, nothing happens. This eighth bit position doesn't activate any pixels.

PIXEL POSITIONS

BINARY

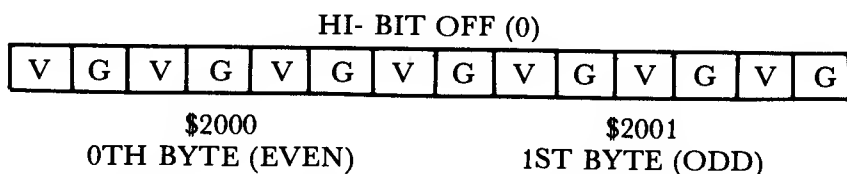
		128	64	32	16	8	4	2	1
	\$01	0	0	0	0	0	0	0	1
	\$02	0	0	0	0	0	0	1	0
	\$04	0	0	0	0	0	1	0	0
	\$07	0	0	0	0	0	1	1	1
	\$08	0	0	0	0	1	0	0	0
	\$0F	0	0	0	0	1	1	1	1
	\$1F	0	0	0	1	1	1	1	1
	\$7F	0	1	1	1	1	1	1	1
	\$80	1	0	0	0	0	0	0	0
	\$FF	1	1	1	1	1	1	1	1

You can see from the diagram that 2000:07 turns on the first three pixels and either 2000:7F (127) or 2000:FF (255) turns on all seven dots. As you shall see shortly, the eight bit, the high bit or most significant bit, is used for color control. While it is not important to use the hi bit in black and white graphics, it does explain why there is a WHITE1 and WHITE2, as well as a BLACK1 and BLACK2. The difference between WHITE1 and WHITE2 is whether or not the hi bit is set.

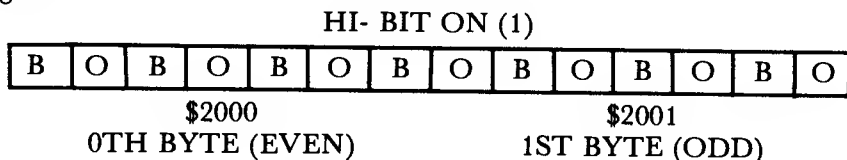
Those using a color TV as a monitor will notice that some of the lit pixels are a violet like color (magenta) while others are green. The Apple II's designers

alternated the colors every other column. The leftmost column in any row always starts with violet if the high bit is off, followed by green in the next column. Thus, there are 140 violet-green pairs in any row. Since the leftmost column is column 0, violet pixels are always in even columns, (i.e., 0,2,4 ... 278). Conversely, green pixels are always in odd columns (i.e. 1,3,5 ... 279).

There is a logical reason for alternating the Apple's colors from column to column. The pairs of colors are related to the square wave pulses in respect to the colorburst reference signal in television receivers. If the Apple sends a pulse that corresponds with the peak of the color signal, you get one color; if the pulse corresponds to the low point of the color signal, you get the complementary color. The Apple can send a pulse shifted 1/4 cycle (in between). That generates two other complementary colors, also in adjacent pairs. I should note that this arrangement is completely independent of the physical locations of the colored phosphors on the television picture tube.



When the hi-bit is set in any byte, the pixel colors shift to blue (cyan) and orange.



When color is considered, there are three primary colors; green, blue and red. Each primary color has a complement. These are magenta (violet), yellow, and cyan (blue) respectively. If a primary color plus its complement are projected on a screen, the result is white, as shown:

PRIMARY COLOR		SECONDARY COLOR	
GREEN	+	MAGENTA (VIOLET)	= WHITE
BLUE	+	YELLOW	= WHITE
RED	+	CYAN (LIGHT BLUE)	= WHITE

What happens on a color monitor is quite similar. If only the first pixel is lit, you get a violet dot. If only the second pixel is lit, you get a green dot. If the first and second pixels are lit, the colors cancel each other and you get an elongated white dot, which is actually two dots wide. The same is true with the blue-orange pairs, except the hi bit is set.

If you want to draw a solid line of one color over the length of the byte, you must turn on the correct sequence of bits.

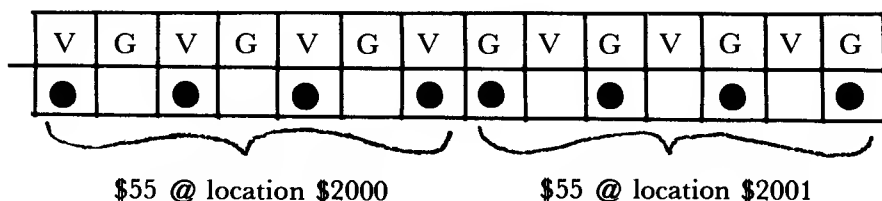
V/B	G/O	V/B	G/O	V/B	G/O	V/B	HI-BIT		
							OPT	\$00 or \$80	BLACK
●		●		●		●		\$55	VIOLET
	●		●		●			\$2A	GREEN
●		●		●		●	●	\$D5	BLUE
	●		●		●		●	\$AA	ORANGE
●	●	●	●	●	●	●	OPT	\$7F or \$FF	WHITE
1	2	4	8	16	32	64	128	VALUE (DECIMAL)	

EVEN BYTE

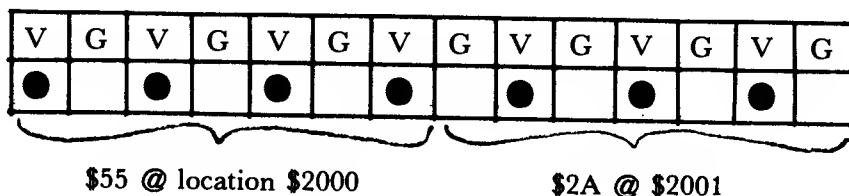
One of the first things you notice, is that although violet and green pixels can be mixed in the same byte, violet and orange pixels can't. The hi-bit is either on or off. You must settle for combinations of violet and green, or blue and orange.

Applesoft users might recall some of the color problems they have encountered in the past. If you were plotting an orange horizontal line starting at column 0 that extended some 20 pixels across the screen and then attempted to plot a white line vertically in column 0 that crossed that orange line, the first few pixels would suddenly turn green. This is because the white color chosen, WHITE1, turned the hi bit off.

The unfortunate result in choosing seven pixels per byte is that the starting color of every other byte alternates. The even bytes start with violet, while the odd bytes start with green. If you were to poke a \$55 into location \$2000, you would get a violet line. But if you poked \$55 into location \$2001, you would get a green line, as indicated below:



In order to correct this effect, the pixels in the second byte would have to be shifted over one position so that the value of \$2A would produce violet, as shown below. We will continue this discussion later, when we discuss shape tables.



The following table lists the values needed to display solid colored lines:

COLOR	EVEN OFFSET	ODD OFFSET
VIOLET	\$55	\$2A
GREEN	\$2A	\$55
BLUE	\$D5	\$AA
ORANGE	\$AA	\$D5
WHITE	\$7F	\$7F
	\$FF	\$FF
BLACK	\$00	\$00
	\$80	\$80

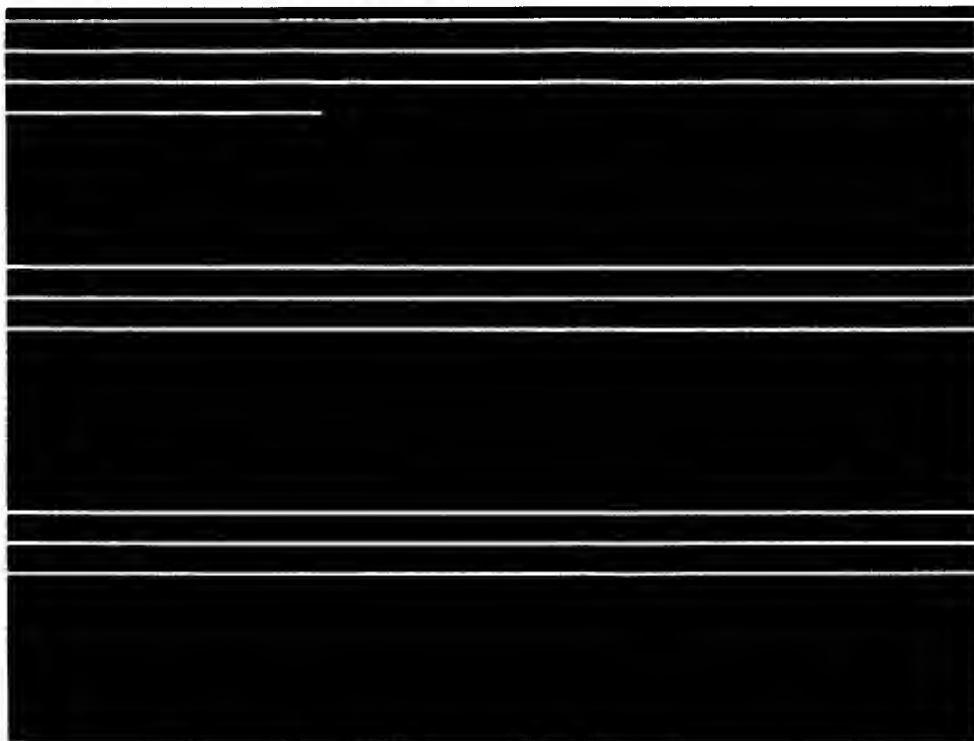
It is an understatement to say that if you were to map the sequential memory locations of the Hi-Res display, they would not map row by row down the screen as you would expect the television's raster scan to plot these pixels. To illustrate this point, let's plot white line segments on a screen by poking a \$FF or decimal 255 into each sequential byte of the Hi-Res page one screen memory.

```

10 HGR : POKE -16302,0
20 FOR I = 8092 TO 16384
30 POKE I,255
40 NEXT I
50 END

```

As you would expect, the computer plotted the first 40 bytes across row 0, but the next 40 bytes appeared 1/3 of the screen below on line 64. The third group of 40 bytes appeared 64 rows below that in the bottom third of the



screen. You would then expect the 4th line to plot directly below line 0 but no, it appears as line eight. Soon the whole display fills up first by thirds, then in groups eight lines apart. If the plotting is stopped with a control C when the screen is half filled, you will notice that there are 24 groups of eight lines.

Perhaps the most frequently asked question about the Hi-Res screen is: Why would the designers make programming the screen so difficult? In 1977, computer components were much more expensive. In an effort to produce a computer for a mere \$1200, several short cuts were taken in the video circuits. Two OR gates were saved by incorporating this strange interlacing with the television's raster scan.

If you look at the memory addresses for the beginning of each of the 192 screen lines, you begin to detect a pattern. The difference in base addresses between any two lines in one of the 24 subgroups is + 1024 bytes, or \$400. The differences between each subgroup in each third of the screen is + 128 bytes. And finally, the difference between lines between each third section is + 40 bytes.

LINE

ADDRESS

1ST SUBGROUP	0	\$2000	8192	} + 1024 BYTES	} + 128 BYTES	
	1	\$2400	9216			
	2	\$2800	10240	} + 128 BYTES		
	3	\$2C00	11264			
	4	\$3000	12288	} + 128 BYTES		
	5	\$3400	13312			
	6	\$3800	14336	} + 128 BYTES		
	7	\$3C00	15360			
	8	\$2080	8320	} + 128 BYTES		
	9	\$2480	9344			
.	.	.	} + 40 BYTES (THIRDS)			
.	.	.				
7TH SUBGROUP	48	\$2300	8960	} + 1024 BYTES	} + 128 BYTES	
	49	\$2700	9984			
	50	\$2B00	11008	} + 128 BYTES		
	51	\$2F00	12032			
	52	\$3300	13056	} + 128 BYTES		
	53	\$3700	14080			
	54	\$3B00	15104	} + 128 BYTES		
	55	\$3F00	16128			
	56	\$2380	9088	} + 1024 BYTES		
	57	\$2780	10112			
8TH SUBGROUP	58	\$2B80	11136	} + 1024 BYTES	} + 128 BYTES	
	59	\$2F80	12160			
	60	\$3380	13184	} + 1024 BYTES		
	61	\$3780	14208			
	62	\$3B80	15232	} + 1024 BYTES		
	63	\$3F80	16256			
	64	\$2028	8232	} + 1024 BYTES		
	.	.	.			
	.	.	.	} + 1024 BYTES		
	.	.	.			
		E T C				

A formula can be derived from the preceding such that, given any line number, the starting memory address for that line can be found. If Y is the line number from 0 to 191, then the section of the screen that the line is in is $A = \text{INT}(Y/64)$. To find which subsection the line is in, use $B = \text{INT}(D/8)$, where $D = Y - 64 * A$. And to find which line Y is on within the subsection, use $C = D - 8 * B$.

$$\text{Memory Location} = 8192 * \text{SN} + 1024 * \text{C} + 128 * \text{B} + 40 * \text{A}$$

where SN = HI-RES PAGE # (1-2).

$$\begin{aligned} \text{Thus, if } Y = 93 \text{ then } A &= \text{INT}(93/64) = 1 \\ D &= 93 - 64 = 29 \\ B &= \text{INT}(29/8) = 3 \\ C &= 29 - 8 * 3 = 5 \end{aligned}$$

If SN = 1 then

$$\text{memory Location} = 8192 + 1024 * 5 + 128 * 3 + 40 * 5 = 13796.$$

An assembly language implementation of this algorithm is shown below.

```

1      *MEMORY ADDRESS FOR START OF SCREEN LINE
2      ORG $6000
3      Y      DS 1
4      A      DS 1
5      D      DS 1
6      B      DS 1
7      C      DS 1
8      TEMP   DS 1
9      SN      DS 1
10     WORKL   DS 1
11     WORKH   DS 1
12     HIRESL   EQU $01
13     HIRESH   EQU HIRESH+$01
6009: AD 00 60 14  START  LDA Y          ;Y=LINE #
600C: 4A      15      LSR              ;DIVIDE BY 32
600D: 4A      16      LSR
600E: 4A      17      LSR
600F: 4A      18      LSR
6010: 4A      19      LSR
6011: 8D 01 60 20  STA  A
6014: 0A      21      ASL              ;MULTIPLY BY 64
6015: 0A      22      ASL
6016: 0A      23      ASL
6017: 0A      24      ASL
6018: 0A      25      ASL
6019: 8D 05 60 26  STA  TEMP          ; TEMP=64*A
601C: AD 00 60 27  LDA  Y
601F: 38      28      SEC              ;SET CARRY TO SUBTRACT
6020: ED 05 60 29  SBC  TEMP
6023: 8D 02 60 30  STA  D              ; D=Y-(64*A)
6026: 4A      31      LSR              ; COMPUTE D/8
6027: 4A      32      LSR
6028: 4A      33      LSR
6029: 8D 03 60 34  STA  B              ; B=INT(D/8)
602C: 0A      35      ASL              ; COMPUTE 8*B
602D: 0A      36      ASL
602E: 0A      37      ASL
602F: 8D 05 60 38  STA  TEMP          ; TEMP=8*B
6032: AD 02 60 39  LDA  D
6035: 38      40      SEC              ;SET CARRY
6036: ED 05 60 41  SBC  TEMP          ;SUBTRACT TEMP
6039: 8D 04 60 42  STA  C              ; C=D-(8*B)

```


603C:	A9 00	43	LDA	#\$00	;CLEAR WORKING REGISTER
603E:	8D 07 60	44	STA	WORKL	
6041:	8D 08 60	45	STA	WORKH	
6044:	AD 06 60	46	LDA	SN	;LOAD SCREEN #
6047:	0A	47	ASL		;MULT BY 32
6048:	0A	48	ASL		
6049:	0A	49	ASL		
604A:	0A	50	ASL		
604B:	0A	51	ASL		
604C:	8D 08 60	52	STA	WORKH	;STORE IN HIGH ORDER
604F:	AD 04 60	53	LDA	C	; LOAD C
6052:	0A	54	ASL		; MULTIPLY BY 4
6053:	0A	55	ASL		
6054:	6D 08 60	56	ADC	WORKH	; ADD TO PREVIOUS HI ORDER
6057:	8D 08 60	57	STA	WORKH	; STORE BACK IN HI ORDER
605A:	AE 03 60	58	LDX	B	; RECALL B
605D:	E8	59	CONT	INX	
605E:	CA	60	DEX		
605F:	FO 14	61	BEQ	SKIPO	; CHECK FOR B=0
6061:	CA	62	DEX		
6062:	FO 0C	63	BEQ	SKIP1	; CHECK FOR B=1
6064:	CA	64	DEX		
6065:	A9 01	65	LDA	#\$01	; ADD 1 TO HIGH ORDER
6067:	6D 08 60	66	ADC	WORKH	
606A:	8D 08 60	67	STA	WORKH	
606D:	4C 5D 60	68	JMP	CONT	; CONTINUE COUNTING
6070:	A9 80	69	SKIP1	LDA	;\$80 ;LOAD ACC WITH 128
6072:	8D 07 60	70	STA	WORKL	; ADD TO LOW ORDER
6075:	AD 01 60	71	SKIPO	LDA	A ; RECALL A
6078:	0A	72	ASL		; MULTIPLY BY 32
6079:	0A	73	ASL		
607A:	0A	74	ASL		
607B:	0A	75	ASL		
607C:	0A	76	ASL		
607D:	6D 07 60	77	ADC	WORKL	; ADD TO LOW ORDER
6080:	8D 07 60	78	STA	WORKL	; STORE BACK IN LOW ORDER
6083:	AD 01 60	79	LDA	A	; RECALL A
6086:	0A	80	ASL		; MULTIPLY BY 8
6087:	0A	81	ASL		
6088:	0A	82	ASL		
6089:	6D 07 60	83	ADC	WORKL	; ADD TO LO ORDER
608C:	8D 07 60	84	STA	WORKL	
608F:	AD 08 60	85	LDA	WORKH	; MOVE RESULTS TO ZERO PAGE
6092:	8D 0A 60	86	STA	HIRESH	
6095:	AD 07 60	87	LDA	WORKL	
6098:	85 01	88	STA	HIRESL	
609A:	60	89	RTS		

--END ASSEMBLY--

This implementation is rather lengthy in that it takes 79 instructions. It was chosen more for its clarity rather than for its speed. Notice that the multiplications are tricky, and that $40 * A$ is split into two easier multiplications, $(8 + 32) * A$. A much faster algorithm, taking only 24 instructions to calculate the screen position for the Yth line, and an additional 18 instructions for the X

offset, is listed in the Programmer's Aid Chip at \$D02E under the label HPOSN. It is also listed under HPOSN in the Applesoft ROM at \$F411. The Y coordinate is placed in the Accumulator, the lo byte of the X coordinate in the X- register, and the hi byte in the Y- register. The screen position is returned in HBASL and HBASH in zero page locations \$26 and \$27, respectively. HMASK is stored in \$30.

I would like to make the point that even 24 instructions is far too many if you are doing fast screen animation. Consider the problem of simply plotting a moving star background for your space game. Twenty stars are scattered about the screen. It takes 480 instructions just to locate the starting memory locations for each line where the star is to be plotted. This doesn't even consider the algorithm needed to decide which pixel in which of 40 bytes on the line needs to be activated. Clearly, a much faster method must be devised. That method is called Table Lookup, and it will be thoroughly discussed in the next chapter.

The X coordinate calculation is much clearer, since the 40 bytes in each line are stored sequentially in memory. Recalling that there are 7 bits per byte times 40 bytes per line gives us 280 bits per line.

Given X, the byte offset is

$$E = \text{INT}(X/7).$$

and the position within the byte is

$$F = X - 7 * E$$

For example, if the X coordinate is 152

$$E = \text{INT}(152/7) = 21 \text{ and } F = 152 - 7 * 21 = 5.$$

So, for the screen coordinate (152,93), the memory location is $13896 + 21 = 13917$, the 5th bit activated.

While the formulas for finding the proper byte and bit positions for the X direction are rather simple; dividing by seven normally requires a complicated divide subroutine. Again, speed is a problem. Although I'll present a complex subroutine below to accomplish the job, it is much faster and simpler to resort to Table Lookup algorithms. Still, it is a matter of trade-offs, using speed versus memory. The tables require 384 bytes plus some code; the subroutine requires only the code.

The subroutine below accepts the X coordinate as a hexadecimal value in the A and X registers. The X register contains the hi byte value. It returns the horizontal byte offset in the Y register and the bit position within that byte in the Accumulator. The theory behind the algorithm is rather simple, but the implementation is complicated because to divide the X position (0-279) by 7 to obtain the horizontal offset is tedious in machine language, in addition to being

complicated by the use of a double precision X value (X values >255 require two bytes).

The division is accomplished by successive subtraction. The idea is subtract 140 to find which half of the screen the point lies, then narrow it to which quarter of the screen. When we have located the position within four bytes, seven is subtracted successively until a zero is crossed. The remainder is the bit position within that screen byte. The hexadecimal plotting value is returned from a table.

```

XCOR LDY  #$00
      DEX          ;TEST IF X COORDINATE >255. X COORDINATE
                        ;WOULD CONTAIN A ONE IF TRUE
      BNZ  XCOR2   ;TEST FOR SPECIAL CASE
      SUB  #$FC    ;SUBTRACTS LARGEST MULTIPLE OF 7 IN 255
      LDY  #$24    ;SET PROVISIONAL QUOTIENT
      BNZ  XCOR8
XCOR2 SEC
      SBC  #$8C    ;LEFT OR RIGHT HALF SCREEN?
      BCC  XCOR3
      LDY  #$14    ;RIGHT HALF, SET QUOTIENT
      BNZ  XCOR4
XCOR3 ADC  #$8C
XCOR4 SEC
      SBC  #$46    ;WHICH QUARTER OF SCREEN
      BCS  XCOR5
      ADC  #$46
      JMP  XCOR6   ;SKIP TO 8THS STAGE
XCOR5 PHA          ;SAVE ACC
      TYA          ;GET QUOTIENT
      CLC
      ADC  #$0A    ;INCREMENT FOR QUARTER
      TAY
      PLA
XCOR6 SEC
      SBC  #$23    ;WHICH 8TH OF SCREEN?
      BCS  XCOR7
      CLC
      ADC  #$23    ;RESTORE DIVIDEND
      JMP  XCOR8
XCOR7 PHA
      TYA
      CLC
      ADC  #$05    ;INCREMENT FOR EIGHTS
      TAY          ;RESTORE QUOTIENT

```

```

        PLA
XCOR8 SEC
        SBC  #$07  ;NOW KEEP SUBTRACTING 7
        BCC  XCOR9 ;UNTIL ZERO IS CROSSED
        INY
        BNZ  XCOR8
XCOR9 CLC
        ADC  #$07  ;RESTORE TO GET REMAINDER
        TAX
        LDA  BITS,X;GET BIT FROM TABLE
        RTS
BITS   HEX   01 02 04 08 10 20 40  ;BIT POSITION TABLE

```

To complete the discussion of the Hi-Res screen's architecture, I'd like to mention what happened to the 512 unused bytes in Hi-Res screen memory. Sequential memory is plotted in lines separated into thirds on the screen. The top line of the bottom third (line #128) uses memory locations 8272 through 8311. It then jumps to the top of the screen, but eight lines down, or line #8. These forty memory locations are 8320 through 8359. Notice there is a gap of eight unused bytes. These unused bytes are at the end of every line in the bottom third of the screen. These 64 lines times 8 bytes accounts for the missing 512 memory locations.

RASTER GRAPHICS

Programmers talk about Raster Graphics and Vector Graphics on the Apple II. In reality, due to the nature of the hardware, vector graphics is a misnomer. Television sets and monitors are raster scanners. Starting at the top of the screen, they scan one line at a time and turn pixels on or off as needed. True vector graphics generators have an electron gun that can move in any direction, so that the beam draws directly between end points.

What is meant by Vector Graphics on the Apple is that a line consisting of a string of pixels is drawn by the television's raster scan. However, raster graphics differs in that entire bytes representing parts of the shape or line are placed into Hi-Res memory locations to obtain a Hi-Res picture. You don't deal in individual pixels per se, but in manipulating Hi-Res shapes a byte at a time. The entire shape is plotted as a block. In some literature, it is referred to as the block shape method.

RASTER SHAPE TABLES (PROS AND CONS)

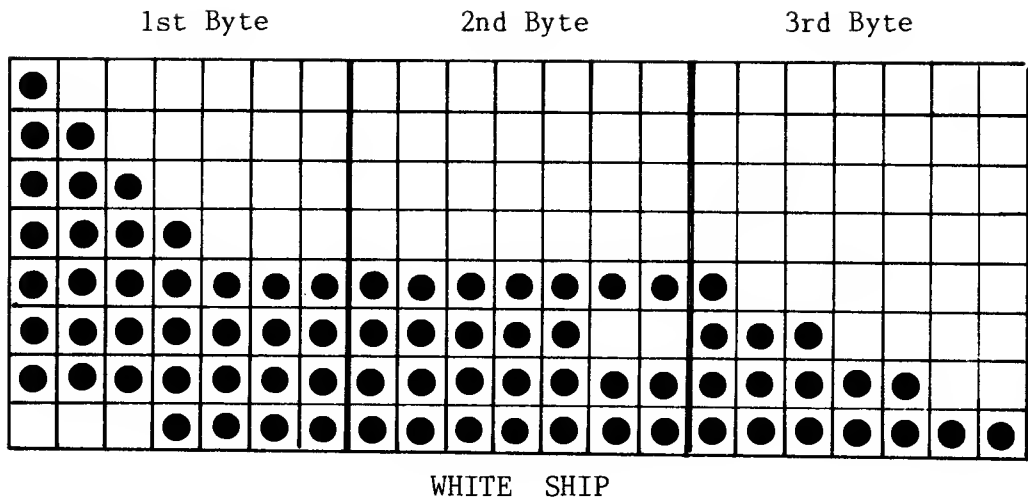
Raster Graphics shape tables, which are bit-mapped shape tables, differ substantially from Apple's Hi-Res shape table routines. Apple's shape table routines, as described in Chapter 1, are plotting vectors that control direction of either plot or no-plot commands. These shape tables can be scaled, rotated, or colored entirely to one of six Hi-Res colors. Bit-mapped shapes, however, are precise instructions used to determine which pixels to activate in a particular section of the screen. Although the shape's detail and color control are superior, they can't be easily scaled or rotated.

At first glance, the pros and cons of using one versus the other appear to be a toss up, but the real advantage in using bit-mapped shape tables is the speed of implementation. Placing a bit-mapped shape table on the screen involves only moving bytes of that table stored in memory to the specific screen memory locations where you want that shape to be drawn. Apple shape tables, on the other hand, require time-consuming machine language routines to translate these plotting vectors into a shape on the screen.

FORMING A BIT MAPPED SHAPE TABLE

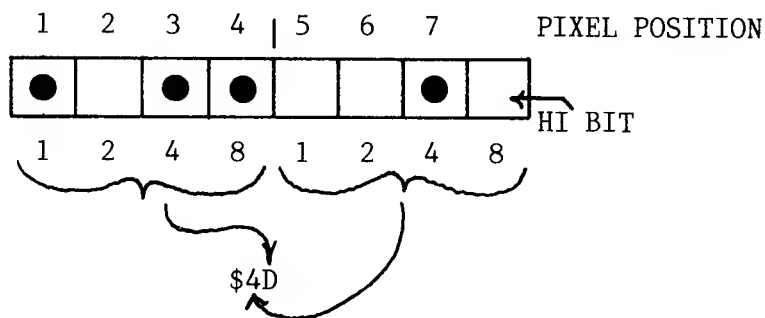
The shape's size must be decided before forming a bit-mapped shape table. A shape can be as large as the entire screen, or as small as one byte wide by one line deep. But in each case, the shape's width is N bytes wide, or a multiple of seven pixels wide. A shape doesn't have to be 7,14,21... pixels wide, but if a shape were, say, 16 pixels wide, it would require a width of 3 bytes. The remaining five pixels would be zeroed.

The second step is to plot the shape's pixels on a sheet of graph paper. A rocket whose shape table can be used later for an arcade game is shown below.

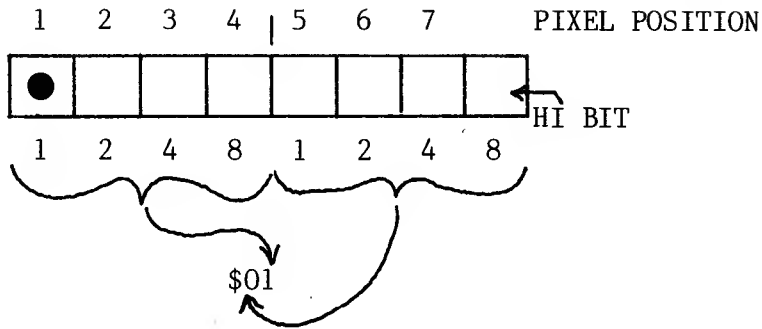


As a first example, we shall plot this shape in white, thus ignoring color problems for the time being. Recall that the color white is produced when adjacent violet and green pixels, or blue and orange pixels, are activated simultaneously. To produce a white ship, all of the pixels will be used to form the table. Some of the readers will question whether the ship is entirely white where bytes have an odd number of pixels, such as in the first and third lines. If you took a magnifying glass to the ship's shape on the TV screen, you would see fringes of violet or green at the edges of an otherwise white ship. This, of course, would not matter on a black and white monitor.

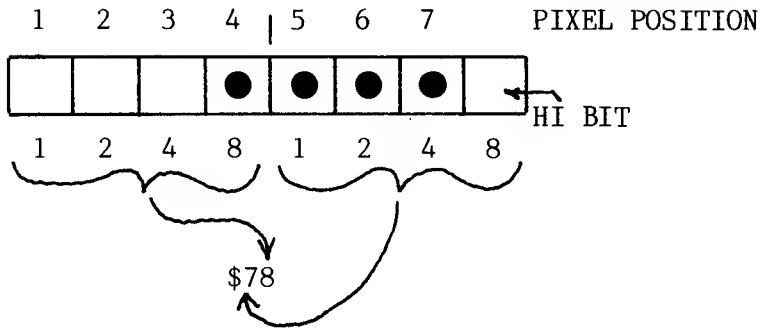
For those that have difficulty converting pixel patterns into hexadecimal values, it is easier if you split the byte's seven bits into a 4-3 pattern. Remember that the right most three dots plus its hi bit is the first part of the byte, or "hi nibble", as four bit halves of a byte are called.



Encoding the rocket's first byte, the first row is as follows:



and the first byte in the last row is:

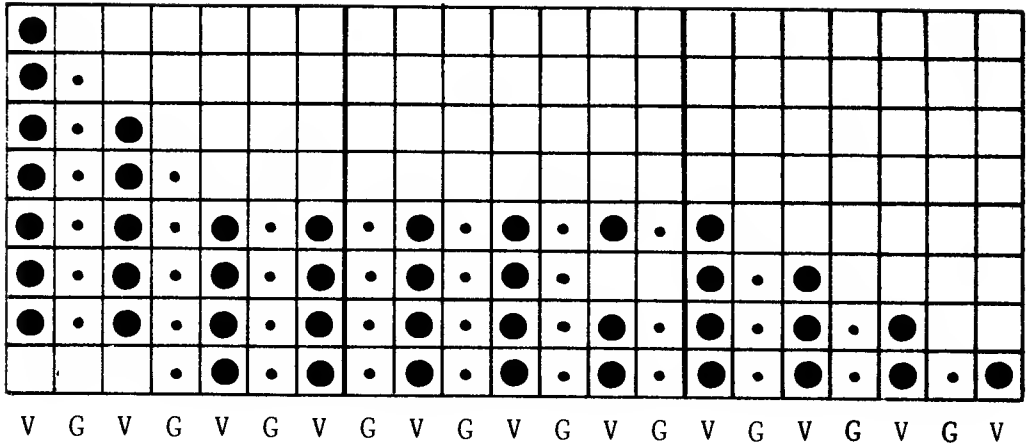


The rocket ship's shape table becomes:

01	00	00
03	00	00
07	00	00
0F	00	00
7F	7F	00
7F	1F	07
7F	7F	1F
78	7F	7F

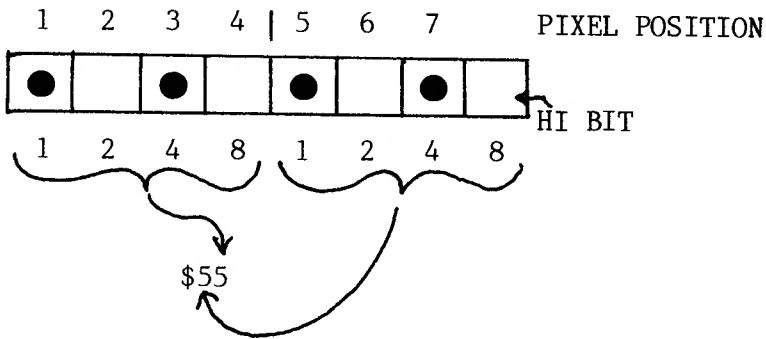
Producing a shape table for the same ship in a particular color presents a more difficult problem. To produce a violet color, all of the green pixels (or those dots in odd columns) must be suppressed. The revised drawing of the ship's shape table is shown below.

VIOLET SHIP (EVEN OFFSET)



where ● — indicates pixel on
 — indicates suppressed dots of original shape

Taking the 5th row, 1st byte as an example:



The complete shape table for the violet colored space ship is:

01	00	00
01	00	00
05	00	00
05	00	00
55	2A	01
55	0A	05
55	2A	15
50	2A	55

At this time it would be instructive to actually plot both white and violet space ships on the Hi-Res screen. This can be done by poking the appropriate bytes into Hi-Res memory.

When we talked about how the screen was mapped, we showed the starting addresses for the first eight lines of the screen. The starting addresses of each line are 1024 bytes or \$0400 apart. Enter the monitor with a CALL -151, then turn on the Hi-Res graphics page 1 and clear the screen as follows:

```
*C050          <CR> ;SET GRAPHICS MODE
*C053          <CR> ;SET MIXED TEXT & GRAPHICS
*C057          <CR> ;SET HI-RES GRAPHICS
*2000:00       <CR>
*2001<2000.3FFM <CR> ;CLEAR PAGE 1 GRAPHICS
```

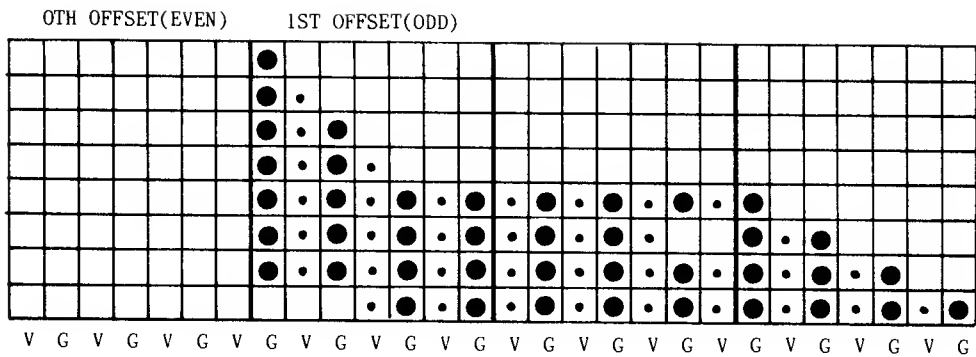
Now poke in the shape table for the white ship. It will appear at the upper left corner of the Hi-Res screen.

```
*2000:01  0000
*2400:03  0000
*2800:07  0000
*2C00:0F  0000
*3000:7F  7F00
*3400:7F  1F07
*3800:7F  7F1F
*3C00:78  7F7F
```

A white ship appears. Now clear the screen and poke in the shape table of the violet ship. The violet ship's table starts at the screen's far left, which is the 0th byte or offset into a particular 40 byte row. Since 0,2,4 are considered even numbers, this is an even offset. As an experiment, poke the violet ship's values into an odd offset, one byte over. First, clear the screen, then type the following:

```
*2001:01  0000
*2400:01  0000
*2800:05  0000
*2C00:.....
      etc.
```

Instead of a violet ship, you get a green space ship. This is because the even offsets start with violet as the first pixel, and the odd offsets start with green. Turning the first pixel on in the odd byte no longer turns on a violet dot, but a green dot. The solution is to use two sets of shape tables; one for even offsets and one for odd offsets. Another solution would be to shift the shape's bit pattern one bit when going from even to odd offsets; however, this is too time consuming for fast animation.



If the original (white) ship's shape is placed so that it begins in an odd offset (above diagram), and the green-columned pixels (the odd columns) are suppressed, the shape becomes:

00	00	00
02	00	00
02	00	00
0A	00	00
2A	55	00
2A	15	02
2A	55	0A
28	55	2A

The first thing that you notice is that the two plotted shapes (even and odd) aren't identical. This can be observed by plotting the even offset table beginning at \$2000, and the odd offset table beginning at \$2005. You will see that the odd offset ship is slightly shorter and the peak of the tail lacks a pixel in row one. This is caused by a lack of symmetry.

This problem can be partially remedied by planning the shape so that the violet column and its adjacent green column are identical in form. For example, if an extra pixel were placed in row 1, column 2 of the original white shape of the ship, the peak of the tail would look identical for both the even and odd offsets.

To reinforce the concept of keeping a shape symmetrical and identical while moving it a byte at a time to the right or left, we will consider the following shape, a green alien:

V G V G V G V G V G V G V G														HEX
EVEN OFFSET (GREEN)			.	●	.	●	.	●						28 01
			.	●	.	●	.	●						28 01
			.	●			.	●						08 01
	●	●			.	●			.	●				22 04
	●	●			.	●			.	●				22 04
	●	●			.	●			.	●				22 04
	●	●			.	●			.	●				22 04
	●	●			.	●			.	●				22 04

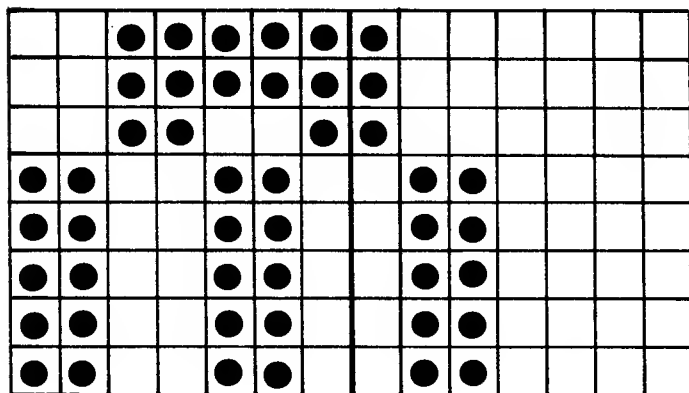
G V G V G V G V G V G V G V														HEX
ODD OFFSET (GREEN)			●	.	●	.	●	.						54 00
			●	.	●	.	●	.						54 00
			●	.			●	.						44 00
	●	.			●	.			●	.				11 02
	●	.			●	.			●	.				11 02
	●	.			●	.			●	.				11 02
	●	.			●	.			●	.				11 02
	●	.			●	.			●	.				11 02

The even and odd offset shapes have been plotted directly below each other to show that the shapes are indeed identical, but the lower shape has been shifted one dot to the left. This effect is inherent in the hardware, because the colors alternate from column to column. Black and white shapes, however, don't require any shifts and, therefore, do not need both odd and even shape tables.

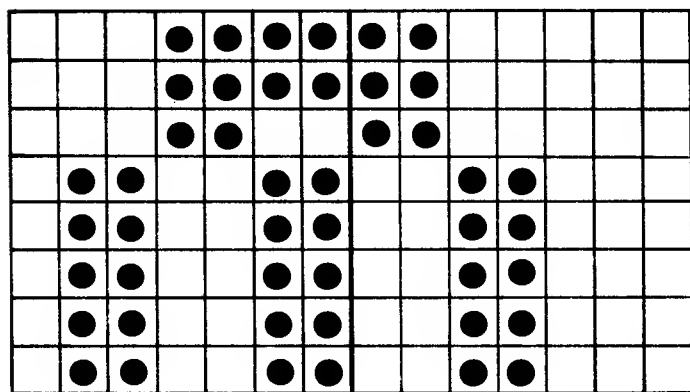
It is important to design your shape with pixels of double width. Otherwise, when you block out the columns of the non-needed color, part of the shape may be absent in the designated color. While this isn't likely to happen if you form shape tables by hand, those ambitious programmers who write a utility to do this automatically might be surprised when plotting their utility generated shape tables.

What we have discussed so far is fine for simply plotting a shape on the screen, or even moving a shape left or right one byte or seven pixels at a time. But what would happen if you wanted to move a shape only one pixel or one horizontal position to the right? If the shape is moved to the right, it no longer has the same bit patterns in each byte.

Consider the alien shape plotted entirely in white. Each time it is shifted right it forms a new bit pattern. By the sixth rightward shift, only the first column of the shape remains in the first byte. Shift it right once more, and we are back to the beginning pattern, but one entire byte to the right.

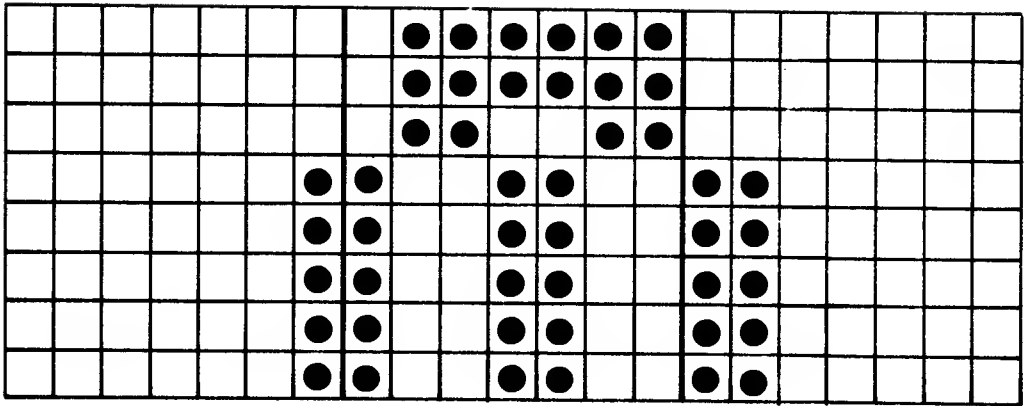


White - 0th Shift



White - 1st Shift

Since the width of a byte is seven pixels, there are seven shifted tables (0-6) for each of the seven positions. When the shape is shifted the fifth time, the pixels extend into a third byte. This requires each of the seven shifted tables to be three bytes wide.



White - 6th Shift

Color shape tables, as you might have guessed, have a similar logic for odd and even offsets. But, as we shall demonstrate, only seven offset tables are needed rather than the expected fourteen.

If you take a simple horizontal line, six pixels wide, as a shape and form a shape table for its green color, you would always have three green pixels lit. As you shift this line over the seven positions, starting first with the even offset, then continuing over the odd offset, you will notice a pattern. Every other time that you shift, the pixel pattern remains the same.

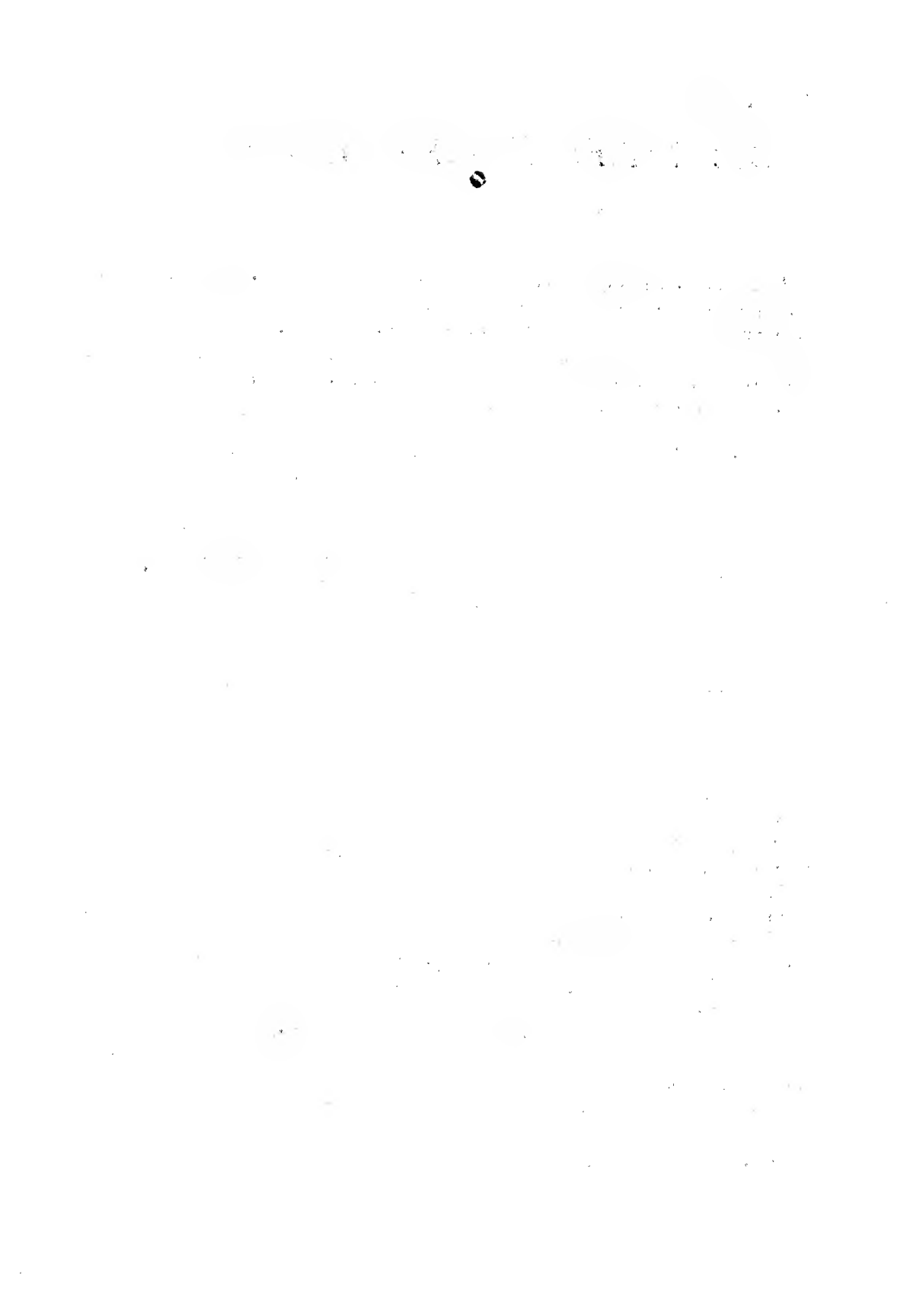
If you were to shift this shape to the right one column for each screen cycle using 14 shape tables, the shape would remain static for two cycles, then move, then stay put for two, then move once again. This produces a very jerky motion. Since the shape tables duplicate themselves in pairs, it would be easier to use the 0th even, 2nd even, 4th even, 6th even, 1st odd, 3rd odd, and 5th odd for a total of 7 shifted tables. The 6th odd shape in the above figure, which appears to be the eighth shape, isn't. It is actually a duplicate of the 0th even shape, but beginning at the next even-odd pair.

In summary you have learned how bit-mapped shape tables are formed. In the next chapter, we shall learn how to draw and animate these shape tables.

EVEN V G V G V G V G V G V G V G V G V G V G V G V

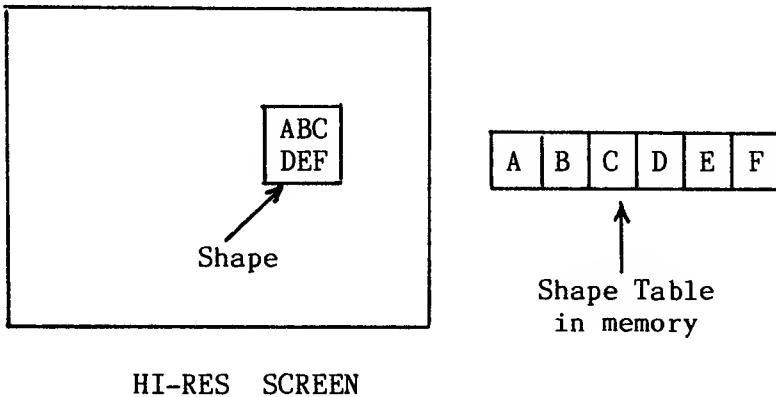
0th				●	.	●	.	●	.													
1st			same	}	.	●	.	●	.	●	.											
2nd				}		●	.	●	.	●	.											
3rd				same	}	.	●	.	●	.	●	.										
4th					}		●	.	●	.	●	.										
5th					same	}	.	●	.	●	.	●	.									
6th						}		●	.	●	.	●	.									

ODD																						
0th								same	}	.	●	.	●	.	●	.						
1st									}		●	.	●	.	●	.						
2nd									same	}	.	●	.	●	.	●	.					
3rd										}		●	.	●	.	●	.					
4th										same	}	.	●	.	●	.	●	.				
5th											}		●	.	●	.	●	.				
6th														.	●	.	●	.	●	.	●	.



BIT MAPPED GRAPHICS

Drawing a bit-mapped shape table anywhere on the Hi-Res screen is a simple procedure once the basic concept is understood. The shape table is stored sequentially in memory, either by rows or by columns. The technique, therefore, is to load each of the bytes, one at a time, into the Accumulator, find the position in memory for the screen location where you want to plot that byte, then store it in that memory location.



The difficulty, as shown in the previous chapter, lies in finding a particular memory location, given an X,Y screen coordinate. Speed is the critical factor in doing arcade animation; therefore, a technique known as Table Lookup is used to locate the starting address of any single line on the Hi-Res screen.

Each of the 192 screen lines has a starting address for the first position (left most) or the 0th offset. The first line or line #0 is located in memory at location \$2000. The second line is at \$2400, etc. Each address takes two bytes. The first part is the hi-byte, which in the later case is \$24. The second byte, \$00, is the lo-byte. These can be separated into two tables, one containing the lower order address of each line (call it YVERTL) and the other containing the higher order address of each line, YVERTH. Each table is 192 bytes long (0-191).

You can access any element in either table by absolute indexed addressing. The effective address of the operand is computed by adding the contents of the Y register to the address in the instruction. That is:

$$\text{EFFECTIVE ADDRESS} = \text{ABSOLUTE ADDRESS} + \text{Y REGISTER.}$$

If our YVERTH table were stored at \$6800 and we wanted to find the starting address of line 1 (remember lines are numbered 0-191), we would index into the table one position and load that value into the Accumulator,

6800:20 24 28 2C 30 34YVERTH TABLE

so LDA YVERTH,Y where Y = \$01 will fetch the value \$24 from memory location \$6800 + \$01 = \$6801, and place it in the Accumulator.

Similarly, if YVERTL were stored immediately after the first table, then:

68C0:00 00 00 00Y VERTL TABLE
Y Register = \$01

LDA YVERTL,Y will take the value \$00 stored in memory location \$68C0 + \$01 = \$68C1, then place it in the Accumulator.

Eventually, we will want to store the first byte from the shape table into memory location \$2400. This can be done efficiently if the two byte address is stored sequentially in zero page. Let's store the lo byte half of the address, HIRESL, at location \$26, and the hi byte half, HIRESH, at location \$27 in zero page:

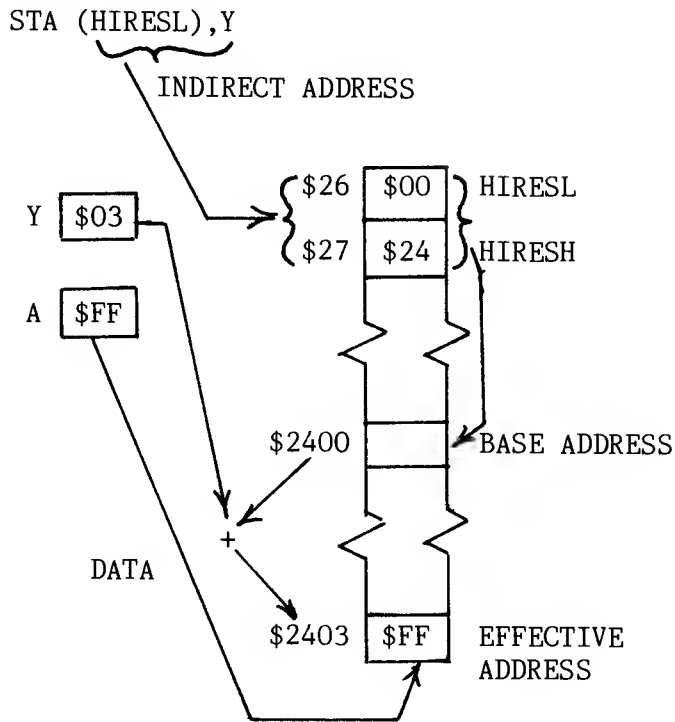
```
LDY  #$01      ;Y REGISTER CONTAINS LINE
LDA  YVERTH,Y  ;LOOKUP HI BYTE OF START
                ;OF ROW IN MEMORY
STA  HIRESH     ;STORE ZERO PAGE
LDA  YVERTL,Y  ;LOOKUP LO BYTE OF ROW IN
                ;MEMORY
STA  HIRESL     ;STORE ZERO PAGE
```

We can change a particular Hi-Res screen memory location using zero page by indirect indexed addressing in the form:

STA (HIRESL),Y Y Reg = \$03

If the computer finds a \$00 in location \$26 (HIRESL) and a \$24 in location \$27 (HIRESH), then the base address is \$2400. The Accumulator stores a value into memory location \$2400 + \$03, or location \$2403, as shown:

INDIRECT INDEXED ADDRESSING

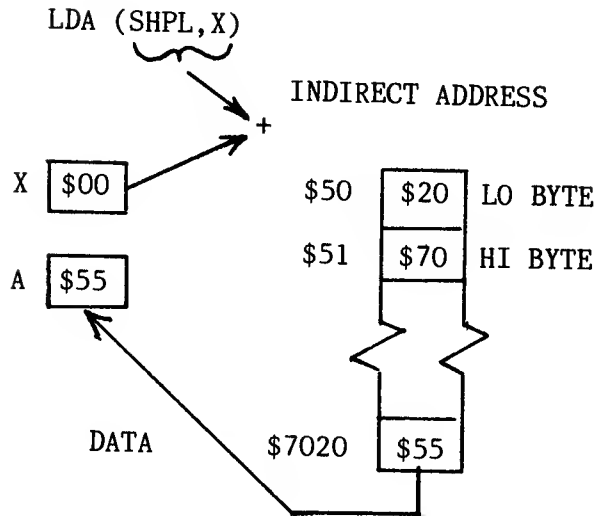


The final addressing mode that we must consider is Indexed Indirect Addressing. It is of the form:

`LDA (SHPL,X)`

It is very similar to the Indirect Indexed addressing mode except the index is added to the zero page base address before it retrieves the effective address. It is primarily used for indexing a table of effective addresses stored in zero page. But in the form we are going to use it, the X register is set to 0; thus, it simply finds a base address:

INDEXED INDIRECT ADDRESSING



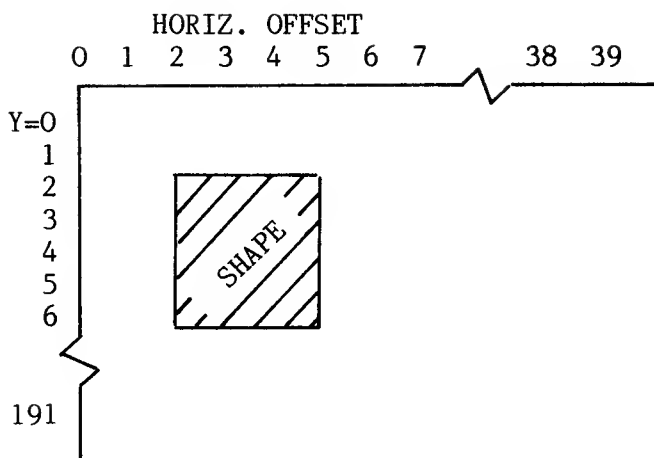
The reason we must use this second form of indirect addressing is a shortage of registers in the 6502 microprocessor. We are already using the Y register in the store operation and there isn't an indirect indexed addressing mode of the form `LDA (SHPL),X`. Thus, we must go to the alternative addressing mode `LDA(SHPL,X)`.

What this all boils down to is that we want to load a byte from a shape table into the Accumulator and store it on the screen with the following instructions:

```
LDA (SHPL,X)    ;STORE BYTE FROM SHAPE TABLE
STA (HIRESL),Y  ;STORE BYTE ON HI-RES SCREEN
```

We can index into the shape table by incrementing the low byte SHPL by one each time, then store that byte into the next screen position on a particular line by incrementing the Y register. This zero page method is faster than doing the equivalent code with absolute index addressing, because two byte addresses can be handled with fewer instructions, less memory space, and with fewer machine cycles.

Obviously, a generalized subroutine must be developed to find the screen memory address (HIRESL & HIRESH), given a line number and a horizontal displacement. We will call this subroutine GETADR, short for Get Address:



Each time a row of shape table bytes is transferred to successive memory locations on the Hi-Res screen, the program will call the subroutine GETADR. The line's starting memory address is then offset by the horizontal location of the shape on the screen.

Memory address = Line # starting address + horizontal offset

```

GETADR  LDA  YVERTL,Y  ;LOOK UP LO BYTE OF LINE
        CLC
        ADC  HORIZ      ;ADD DISPLACEMENT INTO LINE
        STA  HIRESL     ;STORE ZERO PAGE
        LDA  YVERTH,Y  ;LOOK UP HI BYTE OF LINE
        STA  HIRESH
        RTS

```

where the Y register has the vertical screen value (0-191).

If you are designing an arcade game, you will probably have several different shapes on the screen at the same time. Perhaps your defending space ship is paddle-controlled to move vertically but always remains at one particular horizontal offset; while the aliens, attacking in zig-zag fashion, always move horizontally from one side of the screen to the other. Keeping track of each shape's variables, which are inputted into a generalized drawing routine, is more easily done if a setup subroutine is incorporated into your program. This assures that you haven't forgotten to initialize anything before entering the drawing subroutine.

Only a few variables need to be defined in the setup routine: the location of the shape table, the horizontal displacement on the screen, and the width and depth of the shape.

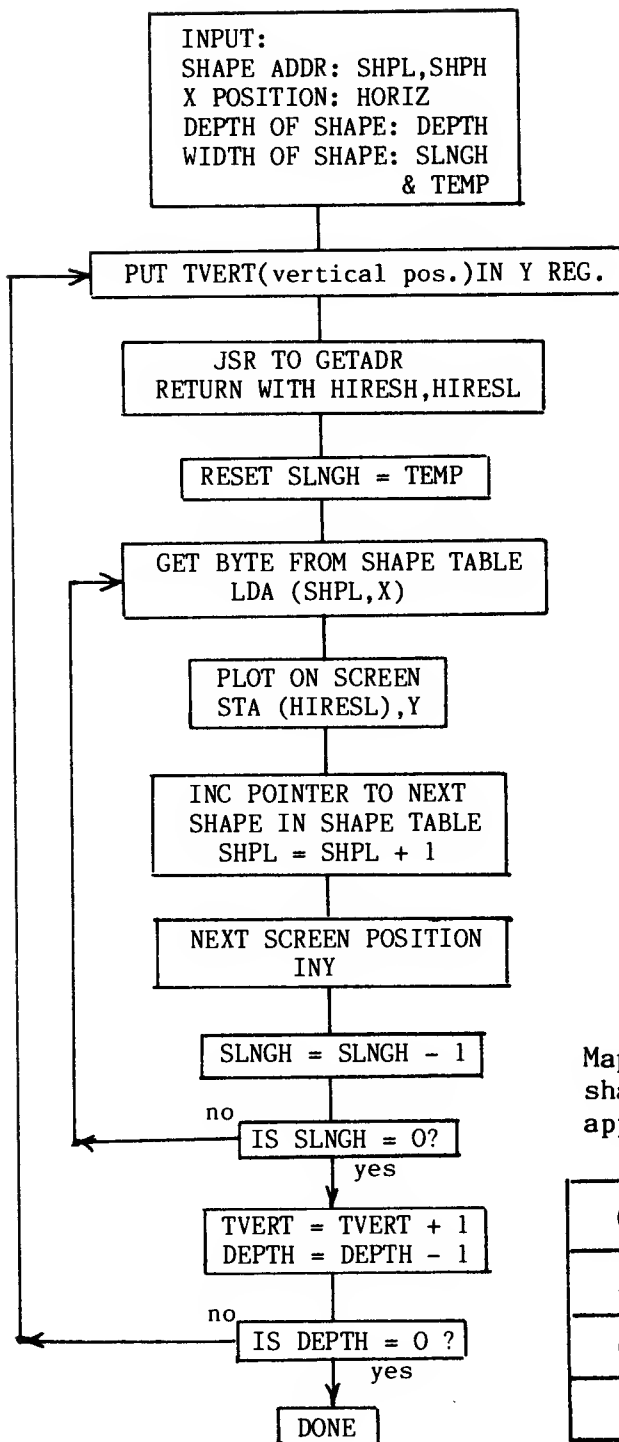
The following example is for the space ship that we designed a shape table for in the last chapter. A word on the notation used for determining the lo and hi addresses for the shape called SHIP is suitable here. In the TED II + and BIG MAC assemblers from CALL APPLE, MERLIN from Southwestern Data Systems, and TOOL KIT from Apple, LDA #<SHIP obtains the lower order address of the table called SHIP. LDA #>SHIP returns the higher order byte of the address. In the LISA assembler from ON-LINE Systems, LDA #SHIP loads the lower order byte and LDA /SHIP loads the higher order byte, as shown:

```
*SHIP SETUP
SSETUP  LDA  #<SHIP  ;LOAD LOWER ORDER BYTE OF SHAPE TABLE
        STA  SHPL
        LDA  #>SHIP  ;LOAD HIGHER ORDER BYTE OF SHAPE TABLE
        STA  SHPH
        LDA  #$08
        STA  DEPTH   ;SHAPE IS 8 LINES DEEP
        LDA  #$09
        STA  HORIZ   ;SHAPE STARTS IN 10TH COLUMN
        LDA  #$03
        STA  SLNGH   ;SHAPE IS 3 BYTES WIDE
        STA  TEMP    ;STORED HERE ALSO BECAUSE DRAWING
                     ;ROUTINE DECREMENTS SLNGH ON EACH
                     ;LINE AND VARIABLE MUST BE RESTORED
                     ;AT START OF NEXT ROW

        RTS
```

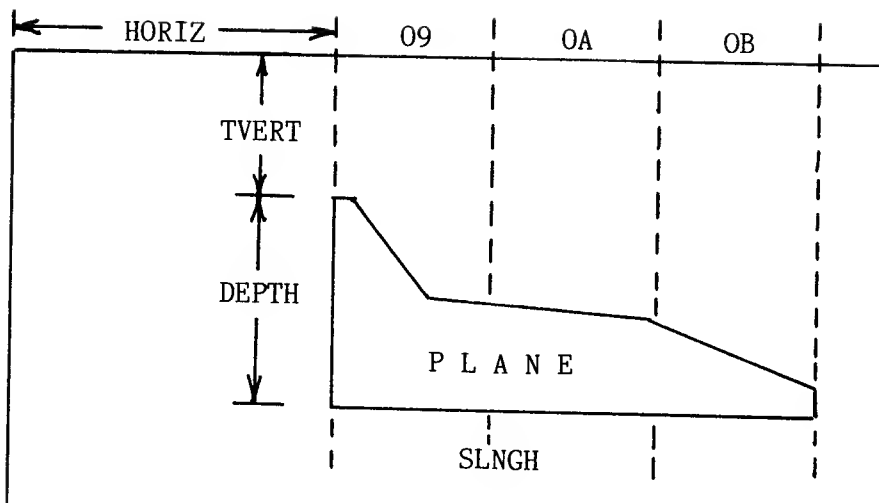
The drawing routine is more efficient the fewer times it accesses the GETADR subroutine. Therefore, it is much faster to load and store on the same screen line until the end of the shape's width is reached. Drawing our spaceship a byte at a time across its width will only require calling GETADR Eight times. But if we plotted down instead, GETADR would be called for each byte, or 24 times, an unnecessary waste of time.

As we load and store across a particular screen line, we decrement SLNGH, the ship's width until SLNGH equals zero. When we are finished with a row, we increment TVERT to the next screen line down and decrement the DEPTH. When DEPTH reaches zero, we have plotted all rows of the shape and we are finished.



Map of elements in
shape table as they
appear on the screen

0	1	2
3	4	5
6	7	8
9	...	



```

DRAW  LDY TVERT          ;VERTICAL POSITION
      JSR GETADR         ;FIND BEGINNING HI-RES SCREEN ADDRESS
                          ;OF ROW

      LDX #$00
      LDA TEMP
      STA SLNGH          ;RESTORE VALUE OF WIDTH FOR NEXT ROW
DRAW2  LDA (SHPL,X)        ;GET BYTE OF SHAPE TABLE
      STA (HIRESL),Y      ;PLOT ON SCREEN
      INC SHPL            ;NEXT BYTE OF SHAPE TABLE
      INY                 ;NEXT POSITION ON SCREEN
      DEC SLNGH           ;DECREMENT WIDTH
      BNE DRAW2           ;FINISHED WITH ROW YET?
      INC TVERT           ;IF SO, INCREMENT TO NEXT LINE
      DEC DEPTH           ;DECREMENT DEPTH
      BNE DRAW            ;FINISHED ALL ROWS?
      RTS                 ;YES, END

```

Although the first row of the shape can be plotted at any TVERT (0-191) position, if TVERT began at 190, the computer would attempt to plot the third line at TVERT, which would equal 192. Indexing into the table that far would most likely produce garbage, as you would index beyond the end of the table. You should be always careful that:

$$TVERT \leq 192 - DEPTH$$

A simple test somewhere before the draw subroutine would suffice. Normally, this should be incorporated into a paddle read-routine. This will be discussed further in the next chapter.

XDRAWING SHAPES

Objects that move on the screen are shifted in position by erasing the object's first position before drawing it at its new position. The simplest method to accomplish this is to draw the shape by exclusive-oring it before shifting it.

The exclusive-or instruction (EOR) is primarily used to determine which bits differ between two operands, but it can also be used to complement selected Accumulator bits. The way it works is elementary. If neither a particular memory bit or Accumulator bit is set or their values are zero, the result is zero. If either one is set, then the result is on. But if both are set, they cancel and the result is zero.

	MEMORY BIT	ACCUMULATOR BIT	RESULT BIT IN ACCUMULATOR
	0	0	0
EOR	0	1	1
	1	0	1
	1	1	0

If we take a byte on the screen and EOR it with the same byte

	0 1 1 0 0 1 1	SHAPE ON SCREEN
EOR	0 1 1 0 0 1 1	SHAPE
	<hr/>	
	0 0 0 0 0 0 0	RESULT

from the shape table, the result is zero or a screen erase. A similar effect would happen if a blank screen were EORed with a shape then EORed once again.

	0 0 0 0 0 0 0	BLANK SCREEN
EOR	0 1 1 0 0 1 1	WITH SHAPE
	<hr/>	
	0 1 1 0 0 1 1	RESULT IS SHAPE ON SCREEN
EOR	0 1 1 0 0 1 1	
	<hr/>	
	0 0 0 0 0 0 0	RESULT IS BLANK SCREEN

Another use for EORing is that it doesn't damage the background if a shape is EORed on the screen, and then off again. However, it does distort the shape slightly.

	0 0 0 0 0 0 1	BACKGROUND
EOR	0 1 0 1 1 0 0	WITH SHAPE
	<hr/> 0 1 0 1 1 0 1	RESULT ON SCREEN (SHAPE
		DISTORTED LAST BIT)
EOR	0 1 0 1 1 0 0	WITH SHAPE
	<hr/> 0 0 0 0 0 0 1	GET BACKGROUND BACK

In the above example, an extra pixel in the shape's last bit position distorts the shape drawn on the screen. In the example below, the fourth bit position becomes a hole in the shape.

	0 0 0 1 0 0 0	BACKGROUND
EOR	0 1 0 1 1 0 0	WITH SHAPE
	<hr/> 0 1 0 0 1 0 0	RESULT ON SCREEN
		hole here
EOR	0 1 0 1 1 0 0	WITH SHAPE
	<hr/> 0 0 0 1 0 0 0	GET BACKGROUND BACK

There are techniques to avoid distorting the shape wherein the background is likely to interfere during the drawing process. This involves a combination of EORing and ORing the Hi-Res screen, with the background stored on a second Hi-Res screen. An alternate method is to store the screen memory bytes in a temporary table equal in size to your shape, while you draw your shape. When erasing, you replace the shape with the background stored in your temporary table. This is a little complicated, but it works. An example using this method is presented at the end of this chapter.

The OR memory with Accumulator (ORA) instruction differs from the EOR instruction in that if both memory and Accumulator bits are on, then the result is one, or on.

	MEMORY BIT	ACCUMULATOR	RESULT BIT IN
		BIT	ACCUMULATOR
	0	0	0
ORA	0	1	1
	1	0	1
	1	1	1

If the background were as follows, and you ORed it with the shape, the shape is correct.

	0 1 0 1 0 1 0	BACKGROUND PAGE 1
ORA	1 1 1 1 0 0 0	WITH SHAPE
	<hr/> 1 1 1 1 0 1 0	GET SHAPE + BACKGROUND WITH NO HOLE IN SHAPE

Unfortunately, if you EOR this result with the shape again, the background is flawed.

	1 1 1 1 0 1 0	SHAPE + BACKGROUND
XOR	1 1 1 1 0 0 0	WITH SHAPE
	<hr/> 0 0 0 0 0 1 0	FLAWED BACKGROUND

Another solution is to take the shape with the background above and EOR it with itself, then EOR it with the background stored on page 2. However, it is probably quicker and easier to just copy the background stored on page 2 directly to screen 1.

	1 1 1 1 0 1 0	SHAPE + BACKGROUND
XOR	1 1 1 1 0 1 0	WITH ITSELF
	<hr/> 0 0 0 0 0 0 0	LOSE EVERYTHING
XOR	0 1 0 1 0 1 0	WITH BACKGROUND STORED PAGE 2
	<hr/> 0 1 0 1 0 1 0	GET BACKGROUND BACK

We can incorporate the exclusive-or instruction in our XDRAW routine. If we EOR the shape we had previously drawn on the screen, nothing remains.

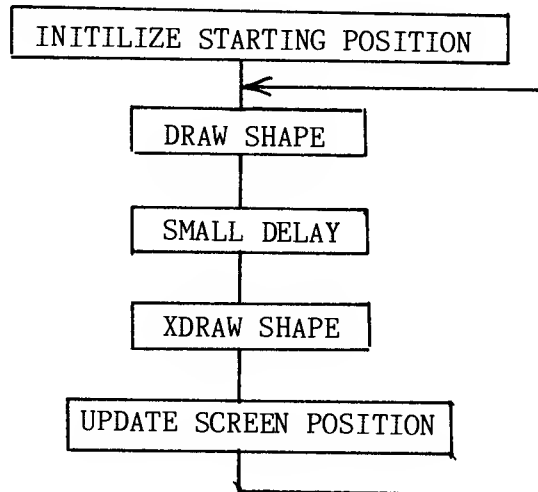
```

XDRAW  LDY  TVERT      ;VERTICAL POSITION
        JSR  GETADR
        LDA  TEMP
        STA  SLNGH      ;RESTORE VALUE OF WIDTH FOR NEXT ROW
        LDX  #$00
XDRAW2  LDA  (SHPL,X)    ;GET BYTE FROM SHAPE TABLE
        EOR  (HIRESL),Y  ;XOR WITH BYTE ALREADY ON THE SCREEN
        STA  (HIRESL),Y  ;DRAW ON SCREEN
        INC  SHPL        ;NEXT BYTE OF SHAPE TABLE

```

INY		;NEXT POSITION ON SCREEN
DEC	SLNGH	;DECREMENT WIDTH
BNE	DRAW2	;FINISHED WITH ROW?
INC	TVERT	;IF SO, INCREMENT TO NEXT LINE
DEC	DEPTH	;DECREMENT DEPTH
BNE	DRAW	;FINISHED ALL ROWS?
RTS		;YES, END ROUTINE

Now that we know how to DRAW and XDRAW a bit-mapped shape anywhere on the Hi-Res screen, the principle for animating these shapes is the same as for Apple shapes discussed previously in Chapter 1. A shape is erased from the screen, its new position is calculated, then it is redrawn at this new position. The procedure is outlined below:



A delay has been inserted between the DRAW and the XDRAW to allow the object to be on the screen longer than it is off. Without the delay, the object is erased immediately after it is drawn. This does not give the shape's image sufficient time to remain on screen during one animation frame. The result is a badly flickering image. The necessary delay can be accomplished by a call to the monitor WAIT subroutine. A hundredth of a second delay is sufficient, but it could be doubled by changing the value in the Accumulator to \$56.

```

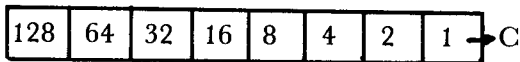
LDA  #$3C
JSR  $FCA8 ;CALL TO WAIT SUBROUTINE

```

COLOR PROBLEMS WITH HORIZONTAL MOVEMENT

When colored shapes are moved vertically, as with our paddle driven space ship, they remain in either the same even or odd offset in which they started. However, when an object moves horizontally a byte at a time, colors shift, or alternate, as the shape moves from an even to an odd offset. As we saw in the last chapter, two different shape tables are needed, one for the even offsets and another for the odd offsets.

An algorithm must be devised to determine whether the HORIZ offset is odd or even. You can ascertain if a value is odd or even by right-shifting the value in the Accumulator so that the low bit enters the carry bit. Since only odd



numbers contain a one in the first bit position, only odd numbers will set the carry. Of course, the carry must be cleared first or this operation will be meaningless.

In order to make the example more meaningful, we will assume we have an even and an odd shape stored in a table called SHAPES. Each shape is one byte wide by eight bytes deep. The even offset shape occupies the first eight bytes, and the odd offset shape follows in the next eight bytes. Let us also assume that the shape table doesn't cross a page boundary (the hi byte is constant).

```

1      *EXAMPLE:COLOR OFFSET PROBLEM & SOLUTION
2      ORG    $6000
3      HORIZ  DS    1
4      SHPL   EQU    $50
5      SHPH   EQU    SHPL+$1
6001: 18      6      CLC                                ;CLEAR CARRY
6002: AD 00 60 7      LDA  HORIZ                        ;LOAD HORIZ VALUE STORED AT $6000
6005: 4A      8      LSR                                ;LOGICAL SHIFT RIGHT INTO CARRY
6006: B0 07      9      BCS  ODD                        ;IF CARRY SET, GOTO ODD CODE
6008: A9 18     10     EVEN  LDA  #<SHAPES              ;LO BYTE OF EVEN SHAPE TABLE
600A: 85 50     11      STA  SHPL
600C: 4C 13 60 12      JMP  CONT
600F: A9 20     13     ODD   LDA  #<SHAPES+8           ;LO BYTE OF ODD SHAPE TABLE
6011: 85 50     14      STA  SHPL
6013: A9 60     15     CONT  LDA  #>SHAPES            ;HI BYTE OF TABLE
6015: 85 51     16      STA  SHPH
6017: 60      17      RTS
18      *
6018: 00 01 02
601B: 03 04 05
601E: 06 07     19     SHAPES  HEX  0001020304050607 ;ODD OFFSET SHAPE
6020: 08 09 0A                                     even
6023: 0B 0C 0D
6026: 0E 0F     20      HEX  08090A0B0C0D0E0F ;ODD OFFSET SHAPE

```

--END ASSEMBLY--

Color shifting problems become more intricate if you intend to do very fine movement or single pixel moves to the left or right, versus coarse movements of a byte or seven pixels at a time. As we discovered in the last chapter, single pixel movements in color aren't effective due to the alternating columns of complementary colors. The shape tends to lag a cycle, then jumps two pixels.

EVEN OFFSET

ODD OFFSET

SCREEN

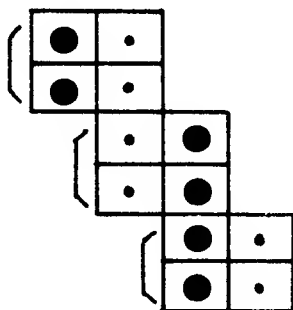
XPOS

0 1 2 3 4 5 6 7 8 9 A B C D

[illegible]

SHAPE TABLE

0	1	2	3	4	5	6
---	---	---	---	---	---	---



The simplest method for keeping track of which offset table is to be used at a particular horizontal position is through tables. One table (XBASE) is needed for the horizontal byte for any horizontal screen position, and another (XOFF) is needed to determine which of the seven offsetted shape table is to be plotted. The tables take the following form:

```

XBASE  HEX 00000000000000
        HEX 00010101010101
        HEX 02020202020202
        HEX 02030303030303

```

```

        .      .
        .      .

```

```

        HEX 26262626262626
        HEX 26272727272727

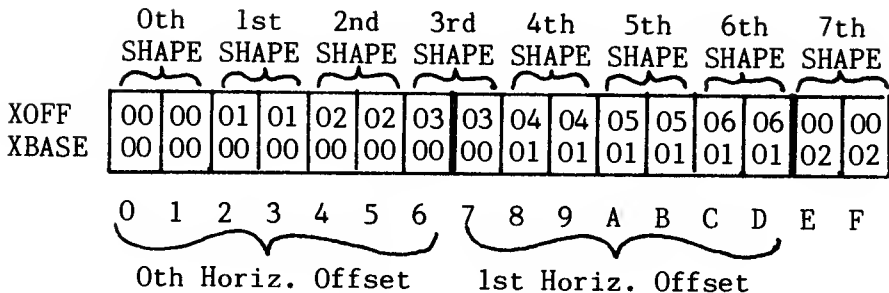
```

```

XOFF   HEX 00000101020203
        HEX 03040405050606
        HEX 00000101020203
        HEX 03040405050606

```

ETC



X COORDINATE VALUE

While the XOFF table is straight-forward in that two adjacent X positions reference the same shape in the table, the XBASE table, which references the horizontal byte offset, requires some explanation. You would assume that all shapes plotted in the first seven horizontal screen positions (X = 0 to 6) would be plotted in the 0th, or even offset, and all shapes plotted in the second seven positions (X = 7 to 13) would be plotted in the first or odd offset. The problem occurs at the boundary of even-odd offset pairs. The third shape table is plotted for both X = 6 and X = 7. But, if the 3rd shape is plotted first in the 0th (even) offset for X = 6, then plotted in the 1st (odd) offset at X = 7, you would get a red shape in the first case, and a blue shape in the second case. The shape would also be shifted over one whole byte, because the shape at X = 7, which is equivalent to that at X = 6 in the odd offset, would instead have an offset of 2; thus it would appear to be at the end of the byte instead of at the beginning.

Therefore, the shape at X = 7 must also be plotted in the 0th (even) offset. I'll be frank and say that the first time I encountered the problem, I spent some time looking for the error by stepping through my code. The solution was that the XBASE tables had to be modified to account for the inconsistency.

The following example will make this clearer. To determine the proper offset and which shape to plot at X = 2, you would calculate as follows:

Look up the third position of XBASE for the offset

or XBASE,2 = \$00

Look up the third position of XOFF for the shape number

or XOFF,2 = \$01

So plot the first shape in 0th offset.

For X = 7

Look up the eighth position of XBASE for the offset

or XBASE,7 = \$00

Look up the eighth position of XOFF for the shape number

or XOFF,7 = \$03

So plot third shape in 0th offset.

This can be formalized into code as part of a setup routine prior to accessing our drawing routine.

```
SETUP LDY  XVALUE
      LDA  XBASE,Y      ;GET BYTE OFFSET FROM TABLE
      STA  HORIZ        ;STORE OFFSET
      LDX  XOFF,Y       ;TABLE TO FIND SHAPE NUMBER
      LDA  SHPLO,X      ;INDEX TO GET LO BYTE OF SHAPE TABLE
      STA  SHPL         ;STORE LO BYTE IN ZERO PAGE
      LDA  #>SHAPES     ;GET HI BYTE OF SHAPE TABLE
      STA  SHPH         ;STORE HI BYTE IN ZERO PAGE
```

SHPLO is a table seven bytes long that contains the lo order byte address of our shapes. Assuming that there are seven shapes, each containing 24 bytes, which are stored at \$800 in a table called SHAPES, then the table takes the following form. The HEX pseudo-op in most assemblers informs the assembler to place hexadecimal data bytes beginning at the location SHPLO. It is equivalent to directly assigning storage space and filling in the values, as follows:

SHPLO HEX 00 18 30 48 60 78 90

OTH 1ST
SHAPE SHAPE ETC.

The obvious intent of the previous method was to save shape table space. If a shape were three bytes wide by eight rows deep, seven tables would require 168 bytes of storage. Requiring the use of all fourteen shapes would double that. While 336 bytes isn't much memory, ten shapes use nearly 3.5K and if any of these were to be rotating shapes, much of memory would be wasted with shape tables.

For those readers who would feel more comfortable calculating and using all fourteen shapes in their table, the code is the same but the tables differ slightly. The tables are more straight-forward because there are no boundary problems.

XBASE HEX 00000000000000
HEX 01010101010101
HEX 02020202020202
.
.
HEX 26262626262626
HEX 27272727272727

XOFF HEX 00010203040506
HEX 0708090A0B0C0D
HEX 00010203040506
HEX 0708090A0B0C0D

SHPLO HEX 00183048607890
HEX A8C0D8F0082038

In this case the shape table extends beyond a page boundary, so a table to reference the Hi byte as well must be included.

SHPHI HEX 08080808080808
HEX 08080808090909

Replace the last two instructions for the hi byte in our setup routine with the following:


```
LDA SHPHI,X ;INDEX TO GET HI BYTE OF SHAPE TABLE
STA SHPH    ;STORE HI BYTE IN ZERO PAGE
```

There is an alternate way to avoid modifying the XBASE table. You could test for the combination of drawing the third shape while at an odd offset.

At first it seemed plausible that using fourteen shape tables might be the better method if, say, the gun were in color and its bullets were in B&W. But since the gun shifted two dots per move, the bullet should do likewise. Besides, the same drawing routines could be accessed.

THE SCREEN ERASE

Erasing an entire Hi-Res screen quickly without the viewer being aware is very important in some games. One well known Asteroid game resorted to a partial (160 line) screen erase instead of XDRAWing the shapes. No one noticed because the frame rate was fast enough, and the animation was page-flipping between graphics screens.

The process is simple and can be used for setting an entire screen to a background color. The Accumulator is loaded with a value (\$00 for black) and stored successively in all 8192 screen memory locations. If we had a sixteen-bit machine and could index all 8192 locations in one gigantic loop, things would be easy. But it has to be done in 256 byte blocks, or in what is called pages of memory. The flow chart is shown below.

Remember that the instruction STA (HIRESL),Y uses a two byte address in zero page

```
$26 = HIRESL = #$00
$27 = HIRESH = #$20
```

then increments it by Y. If Y = \$07, then STA (HIRESL),Y stores what is in the Accumulator in location \$2000 + \$03 = \$2003.

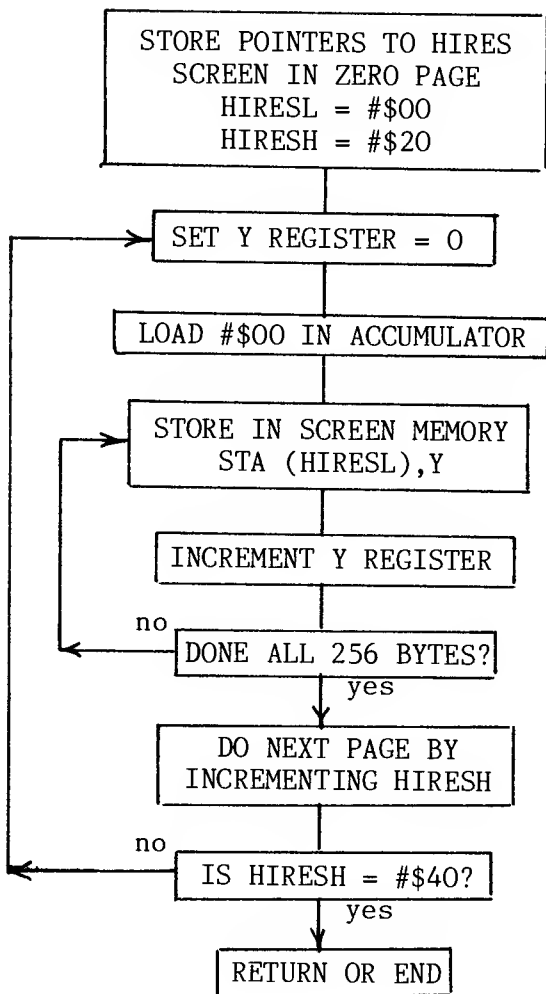
```
HIRESL EQU $26
HIRESH EQU HIRESL +$01

CLRSCR LDA #$00      ;SETUP POINTERS TO CLEAR SCREEN
      STA HIRESL     ;BEGINNING A $2000 (PAGE1)
      LDA #$20
      STA HIRESH
CLR1  LDY #$00      ;PAGE BEGINS AT 0
      LDA #$00      ;LOAD ZERO TO ERASE TO BLACK
CLR2  STA (HIRESL),Y ;STORE IN SCREEN MEMORY
      INY           ;NEXT BYTE
```

```

BNE CLR2      ;DO ALL 256 BYTES; AT 256TH BYTE WRAPS
               ;BACK TO 0 IN Y REGISTER,FALLS THROUGH
INC HIRESH    ;DO NEXT PAGE
LDA HIRESH
CMP #$40      ;FINISHED WITH SCREEN?
BLT CLR1      ;NO, START NEXT 256 BYTE PAGE
RTS           ;YES, ALL DONE

```



This routine takes 35 milliseconds. Note: Screen #2 could be cleared just as easily by storing #\$40 in HIRESH and comparing it to #\$60 to test for the finish.

The screen can be cleared somewhat faster if inline code is used. This is sometimes desirable if part of a screen must be cleared quickly, but becomes a very long and tedious routine if every line is to be cleared. A zero is stored in each screen memory location indicated for a particular column or offset. When it is finished with that column, it increments to the next and clears that, also. Since the code contains the addresses for each line sequentially, precise control can be achieved over what portion of the screen is to be cleared. Of course, other colors can be used too. For instance:

```

        LDA #00          ;BLACK
        LDY #$00         ;START WITH 0TH COLUMN
LOOP    STA $2000,Y       ;ADDRESS OF 0TH LINE
        STA $2400,Y       ;ADDRESS OF 1ST LINE
        STA $2800,Y       ;ADDRESS OF 2ND LINE
        .      .      .
        .      .      . ;Other lines
        INY
        CPY #$28         ;RIGHT SIDE SCREEN?
        BEQ END
        JMP LOOP         ;NEXT COLUMN
END     RTS

```

Sometimes it is desirable to set a Hi-Res screen to a particular color. But color has its inherent odd-even offset problems. For example, to set a screen to blue, a #\$D5 would be stored in all even offset memory locations, while a #\$AA would be required in all odd offset memory locations. Therefore, we have to load and store in pairs as we completely fill the screen memory with bytes that cause only the blue pixels to be activated.

Fortunately, this routine only changes our clear screen routine slightly. You load a #\$D5 for the even offset in the Accumulator, store it at the appropriate screen location referenced by HIRESL & HIRESH, then increment the index or pointer in the Y register. Then #\$AA is loaded and stored for the odd offset in the next screen location. The Y register pointer is then incremented again. Because the BNE test only falls through when the Y register reaches 0 (or actually 256), this can only happen on an even increment. Therefore, the test isn't needed after the first INY, as it can't happen when Y is an odd value.

```

        1      *CLEAR SCREEN COLOR TO BLUE
        2              ORG $6000
        3      HIRESL EQU $26
        4      HIRESH EQU HIRESL+$1
6000: A9 00      5      CLRSCR LDA #$00
6002: 85 26      6              STA HIRESL
6004: A9 20      7              LDA #$20
6006: 85 27      8              STA HIRESH

```

6008:	A0 00	9	CLR1	LDY	#\$00	
600A:	A9 D5	10	CLR2	LDA	#\$D5	;BLUE (EVEN)
600C:	91 26	11		STA	(HIRESL),Y	
600E:	C8	12		INY		
600F:	A9 AA	13		LDA	#\$AA	;BLUE (ODD)
6011:	91 26	14		STA	(HIRESL),Y	
6013:	C8	15		INY		
6014:	D0 F4	16		BNE	CLR2	
6016:	E6 27	17		INC	HIRESH	;DO NEXT PAGE
6018:	A5 27	18		LDA	HIRESH	
601A:	C9 40	19		CMP	#\$40	;FINISHED WITH SCREEN?
601C:	90 EA	20		BCC	CLR1	;NO,START NEXT 256 BYTE PAGE
601E:	60	21		RTS		;YES! DONE

--END ASSEMBLY--

SELECTIVE DRAWING CONTROL & DRAWING MOVEMENT ADVANTAGES

We have seen how background is preserved by EORing shapes on and then off the Hi-Res screen. However, there are times when this is not effective. For instance, complex backgrounds make a mess of a shape, often making it unrecognizable. In these cases, it is best to draw the shape on the screen normally. Naturally, background is lost, but it can be redrawn from memory.

There is another function that is quite important in selective drawing control. That is the And Memory with Accumulator (AND) instruction. It is primarily used to filter or mask out certain bits in the Accumulator or, in the case of the Hi-Res screen, mask out certain pixels. Both the memory bit and the Accumulator bit must be set (on) for the result to be one. If either memory bit or Accumulator bit is off, or both bits are off, the result is zero.

Example:

								Hi bit	
1	0	1	0	1	0	1	1		LDA
0	0	0	0	1	1	1	1		AND
<hr/>									
0	0	0	0	1	0	1	1	RESULT	#\$D0

The above example effectively stripped off the first four pixels of the byte. While it is difficult to design a simple case for using the AND instruction in selective drawing, it is used for "making a hole" in a background before ORing a colored shape into the hole. It is a tricky procedure for beginners, because the complement of an equivalent white shape is used during the AND operation.

We have the following background and colored shape:

1	1	1	1	0	0	0	1	1	0	1	1	1	1	BACKGROUND
1	0	1	0	1	0	1	0	1	0	0	0	0	0	SHAPE

First we need the complement of the white shape.

1	1	1	1	1	1	1	1	1	0	0	0	0	0	WHITE SHAPE CONTAINS
														VIOLET & GREEN
1	1	1	1	1	1	1	1	1	1	1	1	1	1	EOR #\$FF

0	0	0	0	0	0	0	0	0	1	1	1	1	1	
1	1	1	1	0	0	0	1	1	0	1	1	1	1	AND WITH BACKGROUND

0	0	0	0	0	0	0	0	0	0	1	1	1	1	RESULTANT HOLE
---	---	---	---	---	---	---	---	---	---	---	---	---	---	----------------

Now OR the shape into the hole.

0	0	0	0	0	0	0	0	0	0	1	1	1	1	BACKGROUND HOLE
1	0	1	0	1	0	1	0	1	0	0	0	0	0	ORA COLORED SHAPE INTO
														HOLE
1	0	1	0	1	0	1	0	1	0	1	1	1	1	RESULTANT COLORED SHAPE
														& BACKGROUND

Notice that the background doesn't interfere with the colored shape but surrounds it.

The AND instruction is also quite useful in detecting collisions. The procedure will be discussed in detail in the next chapter.

The goal of any programmer is to write fast and efficient code. You can do this by taking advantage of the way the screen is mapped and manipulated in memory. Because it is faster to change a byte, or group of seven pixels rather than each of the pixels separately, it is easier to have separate shapes for each movement to the right or left within a byte. It is also easier to move a shape or object one byte, or seven pixels at a time, horizontally.

Likewise, it is easier during horizontal movement to keep a shape within one of the 24 - eight row subgroups on the Hi-Res screen. If you adhere to that restriction, only the memory address of the first line of the shape need be accessed by tables. Each succeeding line is +\$400 in memory at any given horizontal offset. This method saves many machine cycles by not accessing the GETADR routine for each and every horizontal line in the shape. If your shape is three bytes wide by eight lines deep, the drawing algorithm only has to call the GETADR routine once. Each successive byte in that offset or column is plotted at a location incremented by +\$400 bytes in screen memory. After all

eight bytes have been plotted in that column, screen memory is decremented by \$2000 bytes to return to the top of the subgroup in order to plot in the next column. It is a very fast method, one that many games, like Apple Invaders, uses. If you examine that game, the aliens move slowly across the screen, each character being eight lines deep. When they advance closer to landing, they jump a full eight lines, to be plotted within the next lower eight line subgroup. Although moving 40 aliens may appear slow in the game, there is a very long delay loop. Perhaps some readers have seen the modified version with the hyperspeed option. The game is quite capable of running ten times faster.

The subroutine shown below has the following inputs which can be set in another subroutine called SETUP.

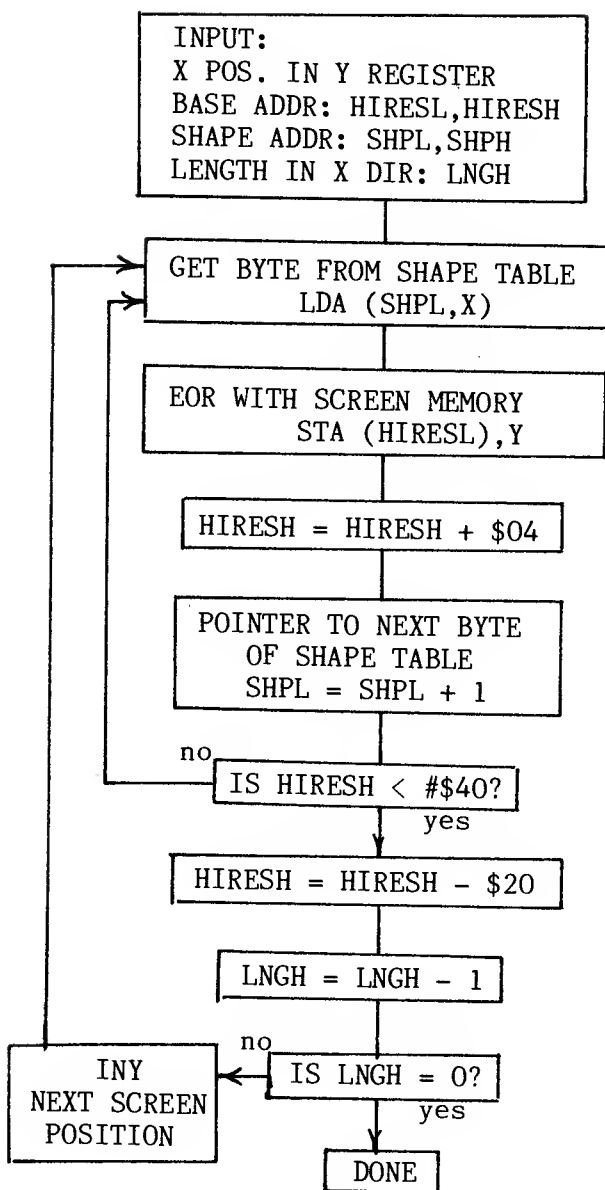
```

*      X POSITION IN Y REGISTER
*      BASE ADDR: HIRESL , HIRESH
*      SHAPE ADDR: SHPL, SHPH
*      LENGTH IN X DIRECTION: LNGH

DRAW  LDX  #$00          ;X-REG MUST BE 0
DRAW2 LDA  (SHPL,X)      ;GET BYTE FROM SHAPE TABLE
      EOR  (HIRESL),Y    ;EXCLUSIVE OR IT WITH WHAT IS ON SCREEN
      STA  (HIRESL),Y    ;PUT IT ON HI-RES SCREEN
      LDA  HIRESH        ;WANT TO REACH NEXT LINE BY ADDING $400
      CLC                ;BY ADDING 4 TO HI BYTE OF BASE ADDR.
      ADC  #$04          ;ADD AFTER CLEARING CARRY
      STA  HIRESH        ;SAVE IT
      INC  SHPL          ;NEXT BYTE OF SHAPE ADDR.
      CMP  #$40          ;ARE WE FINISHED WITH THAT COLUMN
      BCC  DRAW2         ;NO, DO NEXT BYTE
      SBC  #$20          ;YES, BACK TO BASE ADDR (OR TOP)
      STA  HIRESH        ;SAVE IT
      DEC  LNGH          ;NEXT COLUMN SO DECREMENT LENGTH
      BEQ  DRAW3         ;ARE WE FINISHED
      INY                ;DRAW AT NEXT X POSITION
      BNE  DRAW2         ;THIS BRANCH IS ALWAYS TAKEN
DRAW3  RTS              ;DONE!

```

Another way of keeping the code simple is to use only the first 256 horizontal screen positions. This simplifies horizontal paddle routines and eliminates the problem of multi-byte additions to reach screen positions between $X = 256$ and $X = 279$. A large number of games like GAMMA GOBLINS and ASTEROID FIELD have resorted to this technique. The 256 position field need not be left justified, but could be centered using a fixed left margin displacement.



Map of elements in
shape table as they
appear on the screen

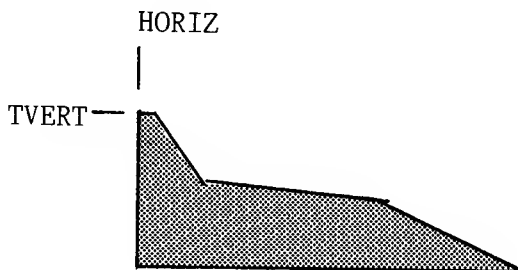
0	8
1	9
2	10
3	.
.	.
.	.
7	15

INTERFACING THE DRAWING ROUTINES TO AN APPLESOFT PROGRAM

Bit-mapped shape tables, as we have seen, are much more detailed and more colorful than APPLE shape tables. There are many programmers not writing a high speed animated game who would like to use these shape drawing routines in an Applesoft program.

If you wanted to control the vertical movement of our space ship by paddle control from an Applesoft program, it can be accomplished in the following manner:

The machine language drawing routine and the setup routine require only the inputs of where to start drawing the ship on the screen. The ship's horizontal location is called **HORIZ** in the machine language subroutine. The ship can be positioned horizontally from the far left (0) to nearly the right hand side of the screen (37). At 37, the ship's nose touches the right screen boundary. Larger values would produce a very strange wrap-a-round, especially at 38 and 39. **HORIZ** is located at \$6001 or 24577 decimal. A value has only to be poked in at this location to change the ship's horizontal location. The ship's vertical position is set by **TVERT**. Its value is trimmed to 0-183 to prevent vertical wrap-a-round. It is located at \$6000 or 24576 decimal. **TVERT** can be directly driven by a paddle routine in the Applesoft program.



The machine language subroutine with code, lookup and shape tables is only 502 bytes long. It starts a \$6006 or 24582 decimal. It sets up the drawing routine before calling it. The drawing routine EOR's the ship's shape to the screen, one byte at a time.

This routine is quite versatile and could handle multiple shapes from Applesoft with little modification to the code. The variables for each shape in the setup routine; lo and hi bytes of the shape, as well as its depth and length, would have to be poked in from Applesoft. The JSR to SSETUP would be removed and the new shapes would be added to the end or in a table elsewhere in memory, in a location where it wouldn't be overwritten by your Applesoft program.

You must be careful with zero page pointers when interfacing BASIC programs to machine language programs. Although I've been lax in choosing locations \$52 through \$58, these conflict with both BASICS. There is a chart in the Apple II Reference manual which shows which zero page locations are free. Safe locations for either BASIC are \$6 to \$9, \$1A to \$1F, \$EB to \$EF, and \$F9 to \$FF. There are others, but I would consult the manual.

Our small Applesoft interface routine is listed below and the machine language code follows.

```

10 HGR: POKE-16302,0           ;SET GRAPHICS
15 H=10 : POKE 24577,H         ;SET HORIZONTAL POSITION
20 TVERT = PDL(1) :IF TVERT >183 ;SET VERTICAL POSITION
    THEN TVERT = 183          WITH PADDLE
25 POKE 24576, TVERT          ;
30 CALL 24582                  ;CALL DRAWING ROUTINE
40 FOR DE = 1 TO 5: NEXT DE   ;SHORT DELAY
45 POKE 24576, TVERT          ;REFRESH VERTICAL POSITION
50 CALL 24582                  ;XDRAW SHIP
60 GOTO 20                     ;LOOP AGAIN

```

```

1      *CODE FOR APPLESOFT PADDLE INTERFACE
2      ORG $6000
3      TVERT DS 1
4      HORIZ DS 1
5      DEPTH DS 1
6      LNCH DS 1
7      SLNGH DS 1
8      TEMP DS 1
9      HIRESL EQU $1A
10     HIRESH EQU HIRESL+$1
11     SSHPL EQU $1C
12     SSHPH EQU SSHPL+$1
13     *MAIN CODE
6006: 20 43 60 14     START JSR SSETUP
6009: 20 0D 60 15     JSR SXDRAW
600C: 60 16     RTS
17     *SUBROUTINES
18     *SHIP DRAWING SUBROUTINE
600D: AC 00 60 19     SXDRAW LDY TVERT ;PADDLE VALUE
6010: 20 2C 60 20     JSR GETADR
6013: A2 00 21     LDX #$00 ;NEED 0 IN X REG. FOR INDEX
6015: A1 1C 22     SXDRAW2 LDA (SSHPL,X) ;LOAD BYTE FROM SHAPE TABLE
6017: 51 1A 23     EOR (HIRESL),Y ;EOR IT AGAINST SCREEN
6019: 91 1A 24     STA (HIRESL),Y ;STORE RESULT ON SCREEN
601B: E6 1C 25     INC SSHPL ;NEXT BYTE IN SHAPE TABLE
601D: C8 26     INY ;NEXT SCREEN POSITION IN ROW
601E: CE 04 60 27     DEC SLNGH ;DECREMENT WIDTH
6021: D0 F2 28     BNE SXDRAW2 ;FINISHED WITH ROW?
6023: EE 00 60 29     INC TVERT ;IF SO, INCREMENT TO NEXT LINE
6026: CE 02 60 30     DEC DEPTH ;DECREMENT ROW
6029: D0 E2 31     BNE SXDRAW ;FINISHED ALL ROWS?
602B: 60 32     RTS

```

602C: B9 5E 60 34	33	*GETADR SUBROUTINE		
602F: 18 35		GETADR LDA YVERTL,Y		;LOOK UP LO BYTE OF LINE
6030: 6D 01 60 36		CLC		
6033: 85 1A 37		ADC HORIZ		;ADD DISPLACEMENT INTO LINE
6035: B9 1E 61 38		STA HIRESL		
6038: 85 1B 39		LDA YVERTH,Y		;LOOK UP HI BYTE OF LINE
603A: AD 05 60 40		STA HIRESH		
603D: 8D 04 60 41		LDA TEMP		
6040: A0 00 42		STA SLNGH		;RESTORE VARIABLE
6042: 60 43		LDY #\$00		
	44	RTS		
6043: A9 DE 45		*SHIP SET UP SUBROUTINE		
6045: 85 1C 46		SSETUP LDA #<SHIP		;LOCATION OF SHIP SHAPE TABLE
6047: A9 61 47		STA SSHPL		
6049: 85 1D 48		LDA #>SHIP		
604B: A9 08 49		STA SSHPH		
604D: 8D 02 60 50		LDA #\$08		;DEPTH 8 LINES
6050: A9 09 51		STA DEPTH		
6052: 8D 01 60 52		LDA #\$09		;STARTING HORIZ POSITION
6055: A9 03 53		STA HORIZ		
6057: 8D 04 60 54		LDA #\$03		;SHIP 3 BYTES WIDE
605A: 8D 05 60 55		STA SLNGH		
605D: 60 56		STA TEMP		
605E: 00 00 00		RTS		
6061: 00 00 00				
6064: 00 00 57		YVERTL HEX	0000000000000000	
6066: 80 80 80				
6069: 80 80 80				
606C: 80 80 58		HEX	8080808080808080	
606E: 00 00 00				
6071: 00 00 00				
6074: 00 00 59		HEX	0000000000000000	
6076: 80 80 80				
6079: 80 80 80				
607C: 80 80 60		HEX	8080808080808080	
607E: 00 00 00				
6081: 00 00 00				
6084: 00 00 61		HEX	0000000000000000	
6086: 80 80 80				
6089: 80 80 80				
608C: 80 80 62		HEX	8080808080808080	
608E: 00 00 00				
6091: 00 00 00				
6094: 00 00 63		HEX	0000000000000000	
6096: 80 80 80				
6099: 80 80 80				
609C: 80 80 64		HEX	8080808080808080	
609E: 28 28 28				
60A1: 28 28 28				
60A4: 28 28 65		HEX	2828282828282828	
60A6: A8 A8 A8				
60A9: A8 A8 A8				
60AC: A8 A8 66		HEX	A8A8A8A8A8A8A8A8	
60AE: 28 28 28				
60B1: 28 28 28				
60B4: 28 28 67		HEX	2828282828282828	
60B6: A8 A8 A8				
60B9: A8 A8 A8				
60BC: A8 A8 68		HEX	A8A8A8A8A8A8A8A8	

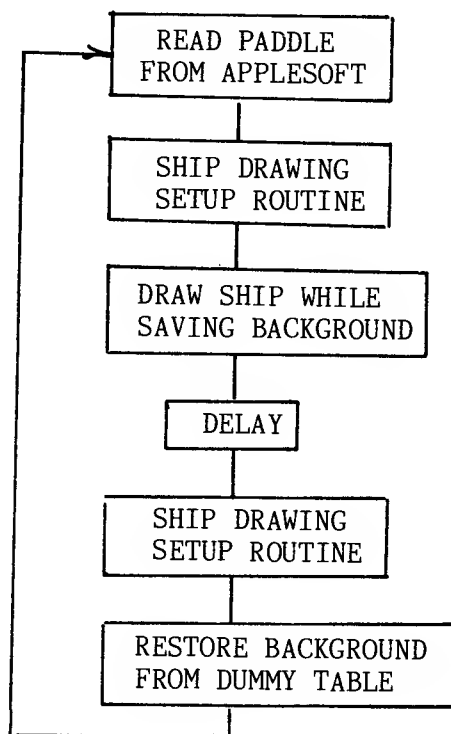
60BE:	28	28	28		
60C1:	28	28	28		
60C4:	28	28		69	HEX 2828282828282828
60C6:	A8	A8	A8		
60C9:	A8	A8	A8		
60CC:	A8	A8		70	HEX A8A8A8A8A8A8A8A8
60CE:	28	28	28		
60D1:	28	28	28		
60D4:	28	28		71	HEX 2828282828282828
60D6:	A8	A8	A8		
60D9:	A8	A8	A8		
60DC:	A8	A8		72	HEX A8A8A8A8A8A8A8A8
60DE:	50	50	50		
60E1:	50	50	50		
60E4:	50	50		73	HEX 5050505050505050
60E6:	DO	DO	DO		
60E9:	DO	DO	DO		
60C:	DO	DO		74	HEX D0D0D0D0D0D0D0D0
60EE:	50	50	50		
60F1:	50	50	50		
60F4:	50	50		75	HEX 5050505050505050
60F6:	DO	DO	DO		
60F9:	DO	DO	DO		
60FC:	DO	DO		76	HEX D0D0D0D0D0D0D0D0
60FE:	50	50	50		
6101:	50	50	50		
6104:	50	50		77	HEX 5050505050505050
6106:	DO	DO	DO		
6109:	DO	DO	DO		
610C:	DO	DO		78	HEX D0D0D0D0D0D0D0D0
610E:	50	50	50		
6111:	50	50	50		
6114:	50	50		79	HEX 5050505050505050
6116:	DO	DO	DO		
6119:	DO	DO	DO		
611C:	DO	DO		80	HEX D0D0D0D0D0D0D0D0
				81	*
611E:	20	24	28		
6121:	2C	30	34		
6124:	38	3C		82	YVERTH HEX 2024282C3034383C
6126:	20	24	28		
6129:	2C	30	34		
612C:	38	3C		83	HEX 2024282C3034383C
612E:	21	25	29		
6131:	2D	31	35		
6134:	39	3D		84	HEX 2125292D3135393D
6136:	21	25	29		
6139:	2D	31	35		
613C:	39	3D		85	HEX 2125292D3135393D
613E:	22	26	2A		
6141:	2E	32	36		
6144:	3A	3E		86	HEX 22262A2E32363A3E
6146:	22	26	2A		
6149:	2E	32	36		
614C:	3A	3E		87	HEX 22262A2E32363A3E
614E:	23	27	2B		
6151:	2F	33	37		
6154:	3B	3F		88	HEX 23272B2F33373B3F
6156:	23	27	2B		
6159:	2F	33	37		

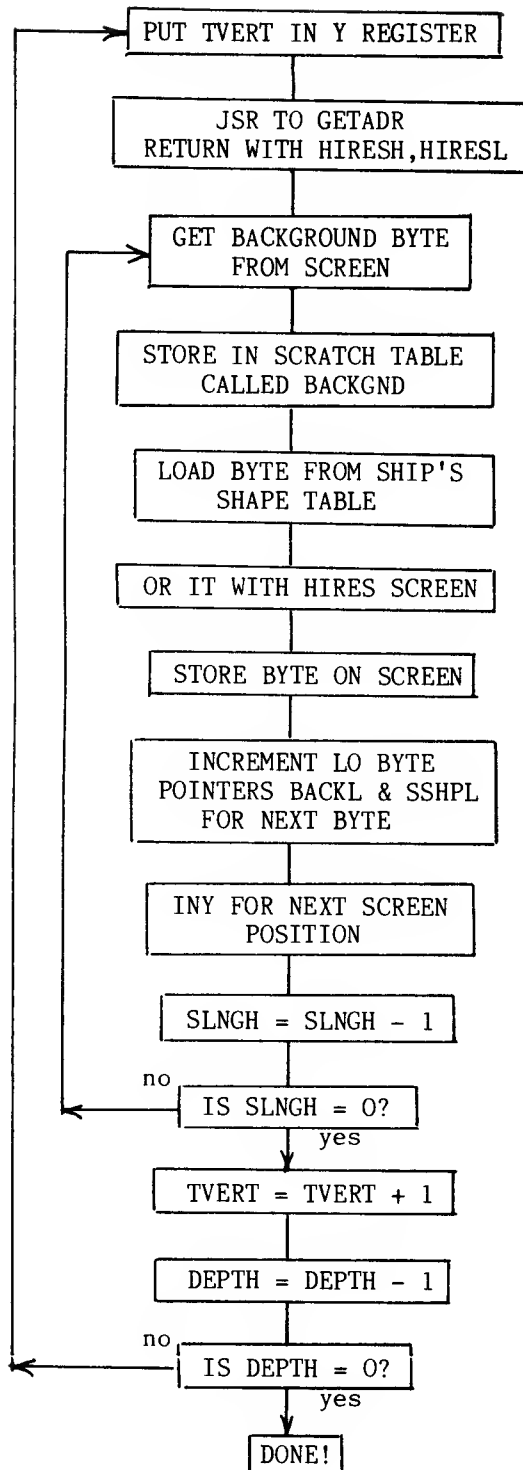
615C: 3B 3F	89	HEX	23272B2F33373B3F
615E: 20 24 28			
6161: 2C 30 34			
6164: 38 3C	90	HEX	2024282C3034383C
6166: 20 24 28			
6169: 2C 30 34			
616C: 38 3C	91	HEX	2024282C3034383C
616E: 21 25 29			
6171: 2D 31 35			
6174: 39 3D	92	HEX	2125292D3135393D
6176: 21 25 29			
6179: 2D 31 35			
617C: 39 3D	93	HEX	2125292D3135393D
617E: 22 26 2A			
6181: 2E 32 36			
6184: 3A 3E	94	HEX	22262A2E32363A3E
6186: 22 26 2A			
6189: 2E 32 36			
618C: 3A 3E	95	HEX	22262A2E32363A3E
618E: 23 27 2B			
6191: 2F 33 37			
6194: 3B 3F	96	HEX	23272B2F33373B3F
6196: 23 27 2B			
6199: 2F 33 37			
619C: 3B 3F	97	HEX	23272B2F33373B3F
619E: 20 24 28			
61A1: 2C 30 34			
61A4: 38 3C	98	HEX	2024282C3034383
61A6: 20 24 28			
61A9: 2C 30 34			
61AC: 38 3C	99	HEX	2024282C3034383C
61AE: 21 25 29			
61B1: 2D 31 35			
61B4: 39 3D	100	HEX	2125292D3135393D
61B6: 21 25 29			
61B9: 2D 31 35			
61BC: 39 3D	101	HEX	2125292D3135393D
61BE: 22 26 2A			
61C1: 2E 32 36			
61C4: 3A 3E	102	HEX	22262A2E32363A3E
61C6: 22 26 2A			
61C9: 2E 32 36			
61CC: 3A 3E	103	HEX	22262A2E32363A3E
61CE: 23 27 2B			
61D1: 2F 33 37			
61D4: 3B 3F	104	HEX	23272B2F33373B3F
61D6: 23 27 2B			
61D9: 2F 33 37			
61DC: 3B 3F	105	HEX	23272B2F33373B3F
61DE: 80 00 00			
61E1: 82 00 00			
61E4: 82 00	106 SHIP	HEX	8000008200008200
61E6: 00 8A 00			
61E9: 00 AA D5			
61EC: 80 AA	107	HEX	008A0000AAD580AA
61EE: 95 82 AA			
61F1: D5 8A A8			
61F4: D5 AA	108	HEX	9582AAD58AA8D5AA

--END ASSEMBLY-- 502 BYTES

When raster or block shapes are plotted against a complex background by EORing them to the screen, the shape is often difficult to discern. As we mentioned in our discussion of the OR function, if a shape is ORed to the screen instead, the shape would be intact. However, this isn't entirely true. The background will affect the shape if either the shape has a window in it, or if true color is always to be preserved. If we had a red locomotive with a black window in the cab and we ORed it against a blue background, the window would not remain black, but would become blue. The color of the train is likely to shift to white because pixels in both the even and odd columns will be activated. A more effective solution would be to AND the complement of a white locomotive shape with the background and then OR the red locomotive to the screen. (See similar example, page 132.)

Background can be saved when ORing a shape to the screen by saving the bytes to a scratch table just before plotting our shape. This is done a byte at a time in sequence with the shape plotting operation rather than as a separate subroutine. Then, when the shape is to be removed from the screen, it isn't XDRAWn; instead, the original background is replotted from this scratch table. I modified the last example to perform this technique and set the background to a color in the Applesoft program so that you could observe the effect. It might be more interesting to load a Hi-Res picture as a very busy background. The code and flow chart are shown below.





```

10 HGR : POKE - 16302,0
12 HCOLOR= 1
13 HPLLOT 100,100: CALL 62454
15 H = 10: POKE 24577,H
20 TVERT = PDL (1): IF TVERT > 183 THEN TVERT = 183
25 POKE 24576,TVERT
30 CALL 24582
40 FOR DE = 1 TO 5: NEXT DE
45 POKE 24576,TVERT
50 CALL 24589
60 GOTO 20

```

```

1      *CODE FOR APPLESOFT PADDLE INTERFACE
2      *WHILE SAVING BACKGROUND
3          ORG $6000
4      TVERT DS 1
5      HORIZ DS 1
6      DEPTH DS 1
7      LNGH DS 1
8      SLNGH DS 1
9      TEMP DS 1
10     HIRESL EQU $1A
11     HIRESH EQU HIRESL+$1
12     SSHPL EQU $1C
13     SSHPH EQU SSHPL+$1
14     BACKL EQU $1E
15     BACKH EQU BACKL+$1
16     *MAIN CODE
6006: 20 6D 60 17     START JSR SSETUP
6009: 20 14 60 18     JSR SDRAW ;DRAW SHIP WHILE SAVING BACKGROUND
600C: 60 19           RTS
600D: 20 6D 60 20     JSR SSETUP
6010: 20 39 60 21     JSR BKDRAW ;REPLACE BACKGROUND
6013: 60 22           RTS
23     *SUBROUTINES
6014: AC 00 60 24     SDRAW LDY TVERT ;PADDLE VALUE
6017: 20 56 60 25     JSR GETADR
601A: A2 00 26         LDX #$00 ;NEED 0 IN X REG. FOR INDEX
601C: B1 1A 27         SDRAW2 LDA (HIRESL),Y ;LOAD BYTE ON SCREEN
601E: 81 1E 28         STA (BACKL,X) ;STORE BACKGROUND TABLE
6020: A1 1C 29         LDA (SSHPL,X) ;LOAD BYTE FROM SHIP SHAPE TABLE
6022: 11 1A 30         ORA (HIRESL),Y ;ORA WITH SCREEN
6024: 91 1A 31         STA (HIRESL),Y ;STOR RESULT ON SCREEN
6026: E6 1E 32         INC BACKL ;NEXT BYTE IN BACKGROUND TABLE
6028: E6 1C 33         INC SSHPL ;NEXT BYTE IN SHIP TABLE
602A: C8 34           INY ;NEXT SCREEN POS. IN ROW
602B: CE 04 60 35     DEC SLNGH ;DECREMENT WIDTH
602E: D0 EC 36         BNE SDRAW2 ;FINISHED WITH ROW?
6030: EE 00 60 37     INC TVERT ;IF SO, INCREMENT TO NEXT LINE
6033: CE 02 60 38     DEC DEPTH ;DECREMENT DEPTH
6036: D0 DC 39         BNE SDRAW ;FINISHED ALL ROWS?
6038: 60 40           RTS ;YES, END ROUTINE

```

6039:	AC	00	60	41	BKDRAW	LDY	TVERT		;PADDLE VALUE
603C:	20	56	60	42		JSR	GETADR		
603F:	A2	00		43		LDX	#\$00		
6041:	A1	1E		44	BKDRAW2	LDA	(BACKL,X)		;LOAD BYTE FROM BACKGROUND TABLE
6043:	91	1A		45		STA	(HIRESL),Y		;STORE ON HIRES SCREEN
6045:	E6	1E		46		INC	BACKL		;NEXT BYTE IN TABLE
6047:	C8			47		INY			;NEXT SCREEN POSITION IN ROW
6048:	CE	04	60	48		DEC	SLNGH		
604B:	DO	F4		49		BNE	BKDRAW2		
604D:	EE	00	60	50		INC	TVERT		
6050:	CE	02	60	51		DEC	DEPTH		
6053:	DO	E4		52		BNE	BKDRAW		
6055:	60			53		RTS			
6056:	B9	90	60	54	GETADR	LDA	YVERTL,Y		;LOOK UP LO BYTE OF LINE
6059:	18			55		CLC			
605A:	6D	01	60	56		ADC	HORIZ		;ADD DISPLACEMENT INTO LINE
605D:	85	1A		57		STA	HIRESL		
605F:	B9	50	61	58		LDA	YVERTH,Y		;LOOK UP HI BYTE OF LINE
6062:	85	1B		59		STA	HIRESH		
6064:	AD	05	60	60		LDA	TEMP		
6067:	8D	04	60	61		STA	SLNGH		;RESTORE VARIABLE
606A:	A0	00		62		LDY	#\$00		
606C:	60			63		RTS			
				64	*SHIP SET UP				
606D:	A9	10		65	SSETUP	LDA	#<SHIP		;LOCATION OF SHIP SHAPE TABLE
606F:	85	1C		66		STA	SSHPL		
6071:	A9	62		67		LDA	#>SHIP		
6073:	85	1D		68		STA	SSHPL		
6075:	A9	28		69		LDA	#<BACKGRD		;LOCATION OF BACKGROUND TABLE
6077:	85	1E		70		STA	BACKL		
6079:	A9	62		71		LDA	#>BACKGRD		
607B:	85	1F		72		STA	BACKH		
607D:	A9	08		73		LDA	#\$08		;DEPTH OF SHAPE
607F:	8D	02	60	74		STA	DEPTH		
6082:	A9	09		75		LDA	#\$09		;STARTING HORIZ. POSITION
6084:	8D	01	60	76		STA	HORIZ		
6087:	A9	03		77		LDA	#\$03		;SHIP 3 BYTES WIDE
6089:	8D	04	60	78		STA	SLNGH		
608C:	8D	05	60	79		STA	TEMP		
608F:	60			80		RTS			
6090:	00	00		00					
6093:	00	00		00					
6096:	00	00	81		YVERTL	HEX	0000000000000000		
6098:	80	80	80						
609B:	80	80	80						
609E:	80	80	82			HEX	8080808080808080		
60A0:	00	00	00						
60A3:	00	00	00						
60A6:	00	00	83			HEX	0000000000000000		
60A8:	80	80	80						
60AB:	80	80	80						
60AE:	80	80	84			HEX	8080808080808080		
60B0:	00	00	00						
60B3:	00	00	00						
60B6:	00	00	85			HEX	0000000000000000		
60B8:	80	80	80						
60BB:	80	80	80						
60BE:	80	80	86			HEX	8080808080808080		
60C0:	00	00	00						
60C3:	00	00	00						

60C6:	00 00	87	HEX	0000000000000000
60C8:	80 80 80			
60CB:	80 80 80			
60CE:	80 80	88	HEX	8080808080808080
60D0:	28 28 28			
60D3:	28 28 28			
60D6:	28 28	89	HEX	2828282828282828
60D8:	A8 A8 A8			
60DB:	A8 A8 A8			
60DE:	A8 A8	90	HEX	A8A8A8A8A8A8A8A8
60E0:	28 28 28			
60E3:	28 28 28			
60E6:	28 28	91	HEX	2828282828282828
60E8:	A8 A8 A8			
60EB:	A8 A8 A8			
60EE:	A8 A8	92	HEX	A8A8A8A8A8A8A8A8
60F0:	28 28 28			
60F3:	28 28 28			
60F6:	28 28	93	HEX	2828282828282828
60F8:	A8 A8 A8			
60FB:	A8 A8 A8			
60FE:	A8 A8	94	HEX	A8A8A8A8A8A8A8A8
6100:	28 28 28			
6103:	28 28 28			
6106:	28 28	95	HEX	2828282828282828
6108:	A8 A8 A8			
610B:	A8 A8 A8			
610E:	A8 A8	96	HEX	A8A8A8A8A8A8A8A8
6110:	50 50 50			
6113:	50 50 50			
6116:	50 50	97	HEX	5050505050505050
6118:	D0 D0 D0			
611B:	D0 D0 D0			
611E:	D0 D0	98	HEX	D0D0D0D0D0D0D0D0
6120:	50 50 50			
6123:	50 50 50			
6126:	50 50	99	HEX	5050505050505050
6128:	D0 D0 D0			
612B:	D0 D0 D0			
612E:	D0 D0	100	HEX	D0D0D0D0D0D0D0D0
6130:	50 50 50			
6133:	50 50 50			
6136:	50 50	101	HEX	5050505050505050
6138:	D0 D0 D0			
613B:	D0 D0 D0			
613E:	D0 D0	102	HEX	D0D0D0D0D0D0D0D0
6140:	50 50 50			
6143:	50 50 50			
6146:	50 50	103	HEX	5050505050505050
6148:	D0 D0 D0			
614B:	D0 D0 D0			
614E:	D0 D0	104	HEX	D0D0D0D0D0D0D0D0
		105	*	
6150:	20 24 28			
6153:	2C 30 34			
6156:	38 3C	106	YVERTH	HEX 2024282C3034383C
6158:	20 24 28			
615B:	2C 30 34			
615E:	38 3C	107		HEX 2024282C3034383C
6160:	21 25 29			

6163: 2D 31 35	
6166: 39 3D 108	HEX 2125292D3135393D
6168: 21 25 29	
616B: 2D 31 35	
616E: 39 3D 109	HEX 2125292D3135393D
6170: 22 26 2A	
6173: 2E 32 36	
6176: 3A 3E 110	HEX 22262A2E32363A3E
6178: 22 26 2A	
617B: 2E 32 36	
617E: 3A 3E 111	HEX 22262A2E32363A3E
6180: 23 27 2B	
6183: 2F 33 37	
6186: 3B 3F 112	HEX 23272B2F33373B3F
6188: 23 27 2B	
618B: 2F 33 37	
618E: 3B 3F 113	HEX 23272B2F33373B3F
6190: 20 24 28	
6193: 2C 30 34	
6196: 38 3C 114	HEX 2024282C3034383C
6198: 20 24 28	
619B: 2C 30 34	
619E: 38 3C 115	HEX 2024282C3034383C
61A0: 21 25 29	
61A3: 2D 31 35	
61A6: 39 3D 116	HEX 2125292D3135393D
61A8: 21 25 29	
61AB: 2D 31 35	
61AE: 39 3D 117	HEX 2125292D3135393D
61B0: 22 26 2A	
61B3: 2E 32 36	
61B6: 3A 3E 118	HEX 22262A2E32363A3E
61B8: 22 26 2A	
61BB: 2E 32 36	
61BE: 3A 3E 119	HEX 22262A2E32363A3E
61C0: 23 27 2B	
61C3: 2F 33 37	
61C6: 3B 3F 120	HEX 23272B2F33373B3F
61C8: 23 27 2B	
61CB: 2F 33 37	
61CE: 3B 3F 121	HEX 23272B2F33373B3F
61D0: 20 24 28	
61D3: 2C 30 34	
61D6: 38 3C 122	HEX 2024282C3034383C
61D8: 20 24 28	
61DB: 2C 30 34	
61DE: 38 3C 123	HEX 2024282C3034383C
61E0: 21 25 29	
61E3: 2D 31 35	
61E6: 39 3D 124	HEX 2125292D3135393D
61E8: 21 25 29	
61EB: 2D 31 35	
61EE: 39 3D 125	HEX 2125292D3135393D
61F0: 22 26 2A	
61F3: 2E 32 36	
61F6: 3A 3E 126	HEX 22262A2E32363A3E
61F8: 22 26 2A	
61FB: 2E 32 36	
61FE: 3A 3E 127	HEX 22262A2E32363A3E
6200: 23 27 2B	

6203:	2F 33 37		
6206:	3B 3F	128	HEX 23272B2F33373B3F
6208:	23 27 2B		
620B:	2F 33 37		
620E:	3B 3F	129	HEX 23272B2F33373B3F
6210:	80 00 00		
6213:	82 00 00		
6216:	82 00	130 SHIP	HEX 8000008200008200
6218:	00 8A 00		
621B:	00 A D5		
621E:	80 AA	131	HEX 008A0000AAD580AA
6220:	95 82 AA		
6223:	D5 8A A8		
6226:	D5 AA	132	HEX 9582AAD58AA8D5A8
		133 BACKGRD DS	24

--END ASSEMBLY--

ERRORS: 0

576 BYTES

ARCADE GRAPHICS

INTRODUCTION

Arcade game animation uses many of the graphics techniques introduced in the previous chapter. Their requirement for high frame rates, coupled with smooth yet detailed animation, necessitates raster shape tables using their inherent high speed drawing routines. Yet, to produce quality games requires game designers to pay particular attention to the smallest programming details.

The fundamentals of any arcade game, in the broad sense, are easy to grasp. It is the details that elude the average programmer. While it is obvious that any object that can be moved must also be controlled, it isn't obvious how that motion is programmed in machine language.

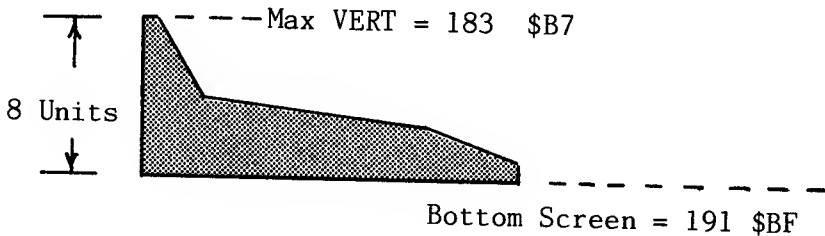
This chapter and the next will discuss the three major types of arcade games and the algorithms that make them work. First, there is the Invaders-type game, wherein a movable gun in the horizontal plane defends against attackers from above. Second, there is the fully maneuverable spaceship from the Space War and Asteroid-type games. These ships fly or float freely in both the X and Y axis. Finally, there are the games that simulate horizontal or vertical motion by scrolling the background. These games have ships that are usually maneuverable in the non-scrolling axis only. Apple games like Pegasus II and Phantoms Five fall into this category.

There are numerous details to consider in game design, such as paddle control, bullets firing and bombs dropping. A game must also include a scorekeeping device for determining a winner, and an explosion subroutine for ridding the screen of losers. And, sometimes, page-flipping techniques are needed to smooth the flickering effects of complex animation. It is hoped that by my first flow charting these routines, then presenting and explaining commented machine language subroutines, you will be able to use these techniques in your own games. And for those who need an example of a working game, many of these routines are combined in a functioning yet unfinished arcade game.

PADDLE ROUTINE

We previously controlled our moveable plane through an Applesoft interface. While it is easy to access the paddle routine directly from machine language, a more realistic subroutine that would prevent almost instantaneous jumps in position needs to be developed. It is the purpose of this section to develop a useable paddle subroutine.

The Hi-Res screen's vertical axis ranges only from 0-191. Paddle values, on the other hand, range from 0-255. An attempt to plot a shape on any horizontal line exceeding 191 would result in unpredictable consequences, because the YVERT tables for the screen address of any line contains only 192 values. Your program might store the shape anywhere in memory, depending on what values might be stored in the locations following our YVERT tables. Therefore, the maximum paddle value can be 191 minus the shape's depth. In the case of our ship, which is eight lines deep, you must clip the paddle value to 183 or \$B7.



A paddle value is read by accessing a monitor subroutine called PREAD, located at \$FB1E. The monitor reads the paddles by writing a strobe to start the selected paddle timer, then increments the Y register until the timer goes off. The paddle value is returned in the Y register. You access PREAD by placing the selected paddle number (0-3) in the X-register. You should be aware that what was previously stored in the Accumulator is destroyed when calling PREAD.

The following paddle subroutine prevents instantaneous jumps of the plane's position by rapid paddle movement. It accomplishes this by adjusting VERT, the ship's vertical position, rather than storing the paddle position (PDL) directly as VERT. This adjustment is based on the relationship of PDL to VERT.

There is a certain maximum paddle-driven movement that is desirable in any game. If the movement, in this case, is set to ten units per frame and the animation was twenty frames per second, then the plane will require approximately one second to move from top to bottom. Slower movement factors will take more time. The speed constant is subjective, and is determined by what you think is a suitable and a controllable speed.

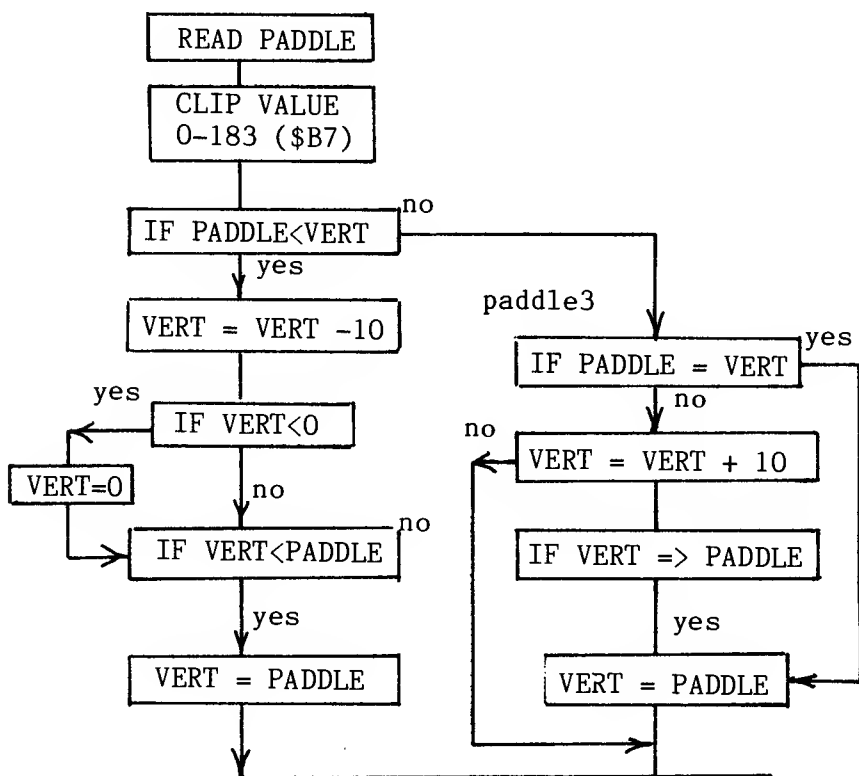
VERT is initialized at 90 decimal to position the ship initially at the center of the screen. If the paddle value is less than VERT, it subtracts ten from VERT and, if greater, adds ten. There are other safeguards to make sure VERT is greater than zero and less than the maximum paddle value, 183 decimal.

There is another test to make sure that VERT actually homes in on the PDL value. Let us assume that VERT was at 70 and the paddle (PDL) is set to 63. Since PDL is less than VERT, ten is subtracted from VERT. VERT is now 60, which is beyond, or less than PDL. But if VERT is less than PDL, it sets

VERT = PDL so that the resulting VERT position is exactly that of the paddle value. The same type of test is performed if PDL is greater than VERT, and VERT is homing in on the paddle value from a higher value.

CYCLE	PDL	VERT		CYCLE	PDL	VERT
0		90	OR	0		90
1	63	80		1	112	100
2	63	70		2	112	100
3	63	63		3	112	112

The flow chart is shown below.

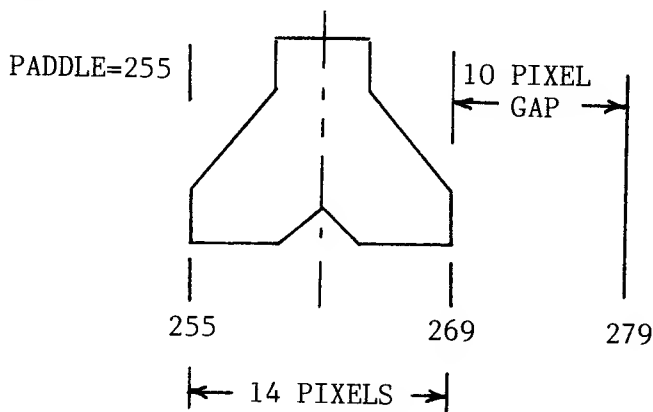


Rather than proceed with the development of what is to become a very complex game using our ship, I would like to digress to another paddle routine. This one controls a moveable gun turret in the horizontal plane. It is used quite frequently in most Invaders-type games.

The screen range on the horizontal axis is 0-279. Our paddle range is, as usual, limited to 0-255. In Applesoft, it was easy to multiply by 1.1 to obtain

the proper range. However, in machine language the multiplication and division routines are too complex, and require numerous machine cycles to execute. Besides, they return the result as two byte values, which means that all of our adding and subtracting would require two byte operations.

It is much easier to accept the fact that the right 10% of the screen is unusable or can't be reached by paddles, unless we center the screen by adjusting the horizontal offsets. Actually, if our gun is large, we can use part of this space without adjustment. Take the gun turret illustrated below. It is 14 pixels, or two bytes wide.



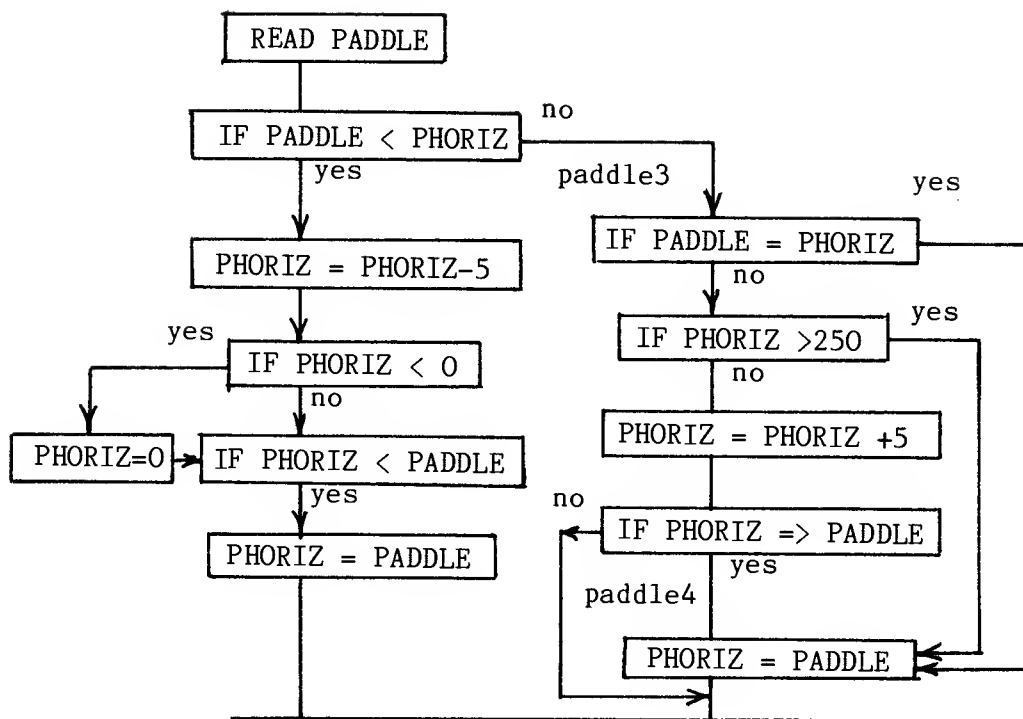
When the paddle value is at zero, the gun plots between 0-13 on the horizontal axis, and when the paddle is at 255, the gun plots between 255 and 269. That leaves only a ten pixel gap, which is hardly noticeable.

In order to use the paddle routine already developed for the vertical axis, it must be modified. The paddle's full range is needed, so clipping is removed just after the paddle is read. Instead, we must place a test in the code to prevent it from incrementing past \$FF (255 decimal) as it homes in on the actual paddle value. In this case, we have slowed the turret's movement to five units per animation cycle. Again, the value of five is based on the frame rate, and what appears to be a reasonable movement rate on the screen.

After testing the various possibilities of whether the paddle is set to a value greater than PHORIZ (the horizontal position) you must prevent it from adding five to PHORIZ if PHORIZ > 250. In this case, the PADDLE value is 251 to 255, and PHORIZ is set equal to the PADDLE.

CYCLE	PADDLE	PHORIZ
2	253	240
1	253	245
2	253	250
3	253	253

The following chart and corresponding code is shown below.



6028:	A2 01	40	*READ PADDLE #1	
602A:	20 1E FB	41	RPDL LDX #\$01	
602D:	8C 07 60	42	JSR PREAD	
6030:	98	43	SKIPP STY PDL	
6031:	CD 0B 60	44	TYA	
6034:	B0 1E	45	CMP PHORIZ	;PADDLE<HORIZ POS THEN SUBTRACT 5
6036:	AD 0B 60	46	BGE PADDLE3	
6039:	38	47	LDA PHORIZ	
603A:	E9 05	48	SEC	
603C:	B0 08	49	SBC #\$05	
603E:	A9 00	50	BGE PADDLE1	;MAKE SURE =>0
6040:	8D 0B 60	51	LDA #\$00	
6043:	8D 0C 60	52	STA PHORIZ	
6046:	CD 07 60	53	STA TPHORIZ	
6049:	B0 03	54	PADDLE1 CMP PDL	;DON'T WANT TO GO PAST PADDLE POS
604B:	AD 07 60	55	BGE PADDLE2	
604E:	8D 0B 60	56	LDA PDL	
6051:	4C 71 60	57	PADDLE2 STA PHORIZ	
6054:	CD 0B 60	58	JMP PADDLE6	
6057:	FO 12	59	PADDLE3 CMP PHORIZ	;PADDLE>PHORIZ POS THEN ADD 5
6059:	AD 0B 60	60	BEQ PADDLE4	
605C:	C9 FA	61	LDA PHORIZ	
605E:	B0 0B	62	CMP #\$FA	;IS PHORIZ>250
6060:	AD 0B 60	63	BGE PADDLE4	
6063:	18	64	LDA PHORIZ	
			CLC	


```

6064: 69 05 65      ADC  #$05
6066: CD 07 60 66      CMP  PDL          ;DON'T WANT TO GO PAST PADDLE POS
6069: 90 03 67      BLT  PADDLE5
606B: AD 07 60 68      PADDLE4 LDA  PDL
606E: 8D 0B 60 69      PADDLE5 STA  PHORIZ
6071: 8D 0C 60 70      PADDLE6 STA  TPHORIZ

```

PADDLE CROSSTALK

Many readers will attempt at some future time to combine two paddle read routines together to control a ship, or a gun crosshair with a joystick. They will be dismayed to learn that the paddle values don't read properly. This is called paddle crosstalk.

When a paddle trigger is strobed, all the timers start. If the first paddle that you read has a low value, it will return quickly from PREAD with a paddle value. But the timers are still counting. If you immediately call PREAD again, the timers aren't restarted at zero, so that you may see a value from the first paddle trigger instead of the second. The solution is to wait a sufficient time before reading the second paddle. How long is sufficient? Not more than 255 machine cycles is needed. It is best to space your paddle reads with other code in between.

An alternate solution is to read two paddles simultaneously by triggering both strobes (or timers) together. Since the code takes longer to execute while the paddle timers count down, the full paddle range can not be expected. The code shown below is suitable for joystick control, but only has a range of 40 to 127. Clever programmers will either adjust these values or offset them to suit their needs.

```

1  *THIS DUAL PADDLE READ RETURNS
2  *VALUES AS FOLLOWS
3  *PADDLE(0),PADDLE(1)
4  *
5  *126,127 -----44,127
6  * !
7  * !
8  * !
9  * !
10 * !
11 * !
12 *126,47 ----- 44,47
13 *
14
15 ZERO      ORG  $300
16 ONE      DS   1
17          DS   1
0302: A2 00      LDX  #$00
0304: 8E 01 03 18 STX  ONE
0307: 8E 00 03 19 STX  ZERO
030A: A2 7F      LDX  #$7F
030C: AD 70 C0 21 LDA  $C070 ;STARTS BOTH TIMERS

```

```

030F: AD 64 C0 22  LOOP    LDA $C064      ;PADDLE 0 TIMER
0312: 29 80 23      AND    #$80
0314: 0A 24         ASL
0315: 2A 25         ROL
0316: 6D 00 03 26    ADC    ZERO
0319: 8D 00 03 27    STA    ZERO
031C: AD 65 C0 28    LDA    $C065      ;PADDLE 1 TIMER
031F: 29 80 29      AND    #$80
0321: 0A 30         ASL
0322: 2A 31         ROL
0323: 6D 01 03 32    ADC    ONE
0326: 8D 01 03 33    STA    ONE
0329: CA 34         DEX
032A: D0 E3 35      BNE    LOOP
032C: A9 7F 36      LDA    #$7F
032E: 38 37         SEC
032F: ED 00 03 38    SBC    ZERO
0332: 8D 00 03 39    STA    ZERO
0335: A9 7F 40      LDA    #$7F
0337: 38 41         SEC
0338: ED 01 03 42    SBC    ONE
033B: 8D 01 03 43    STA    ONE
033E: 60 44         RTS

```

--END ASSEMBLY--

Many game designers choose keyboard controls instead of joystick controls. There are two reasons for this. The first is speed. Obviously, a test for a specific keypress only takes three instructions. A paddle, on the other hand, can take as long as 255 machine cycles. Two paddles (joystick) take nearly twice as long if you avoid crosstalk. There are many games where reading two paddles slows the program down. Several games resort to reading one paddle direction on alternate frames, and the other on the opposite frame; however, the controls seem sluggish. The only sensible solution is to write fast, efficient code, so that reading paddles does not affect the game's speed.

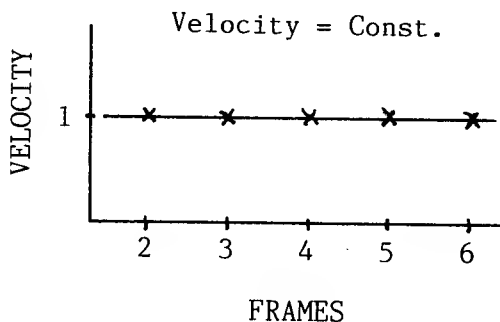
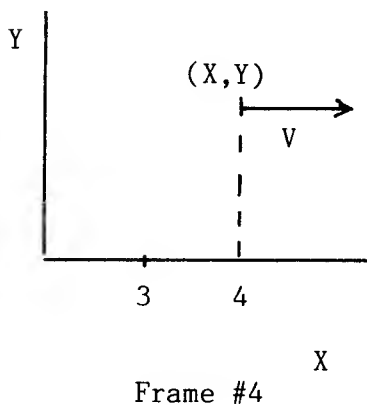
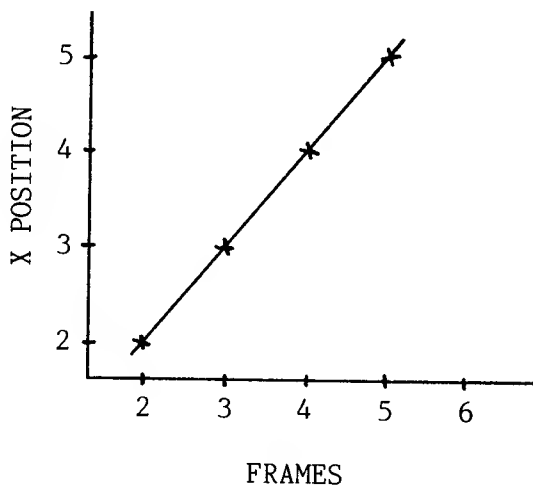
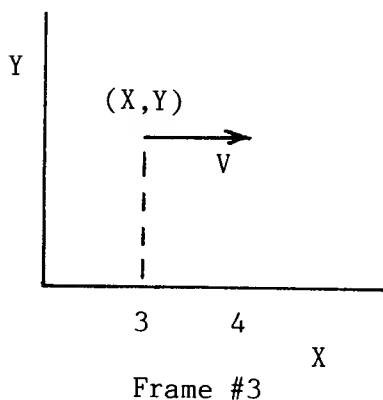
The second reason for keyboard control is that, until recently, few computer owners had joysticks. If the latter is the reason, the designer should offer a choice of control modes. Certainly playability is more important than monetary gain from a wider audience.

DROPPING BOMBS AND SHOOTING BULLETS

Simulating a bomb drop realistically involves some knowledge of how a body in motion reacts to a constant force; in this case, gravity. The physics of a body in motion requires advanced mathematics, mainly calculus. But calculus actually involves the summation of many bits and pieces of a body's velocity and acceleration to determine the actual distance an object travels. The computer, fortunately, automatically divides our time frame into small units, or animation frames, wherein the force vectors can be displayed as direction vectors.

Let's examine an object in simple linear motion. The object is initially at rest. It is then given a horizontal velocity of one unit to the left. Thus, the velocity is $+1$ unit/time frame. During each animation frame, the object moves $+1$ units to the right.

An object's direction of travel and its magnitude is represented by a line segment called a vector. An object's velocity vector always points in the direction of travel. Our object shown below has a velocity of $+1$ units/ time frame, so that the velocity is pointing to the right. Since the velocity vector is to the right, the object moves to the right.



This can be formalized into equations for each of the two screen directions X and Y.

$VX = +1$ velocity is constant in X direction
 $X = X + VX$ new position is the old position plus
 the change in position (velocity).

Likewise

$VY = 0$ velocity is stationary in Y direction.
 $Y = Y + VY$

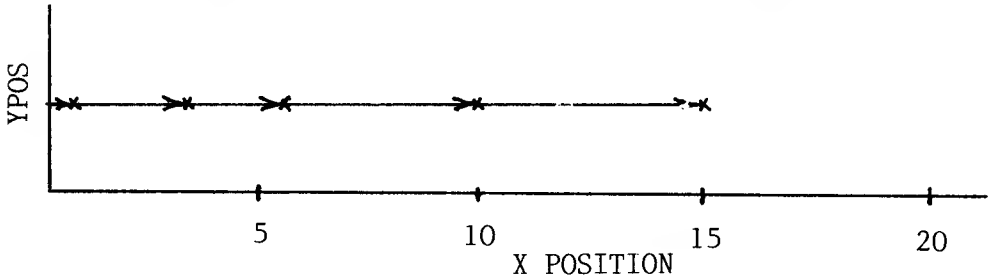
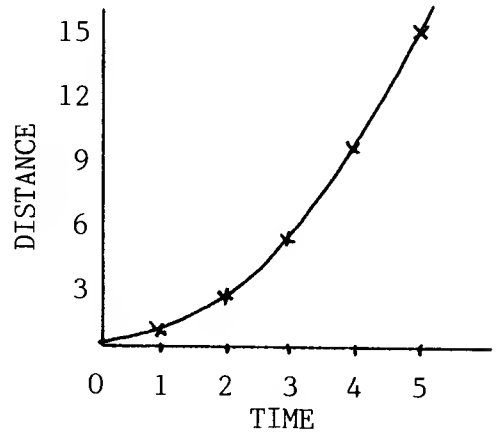
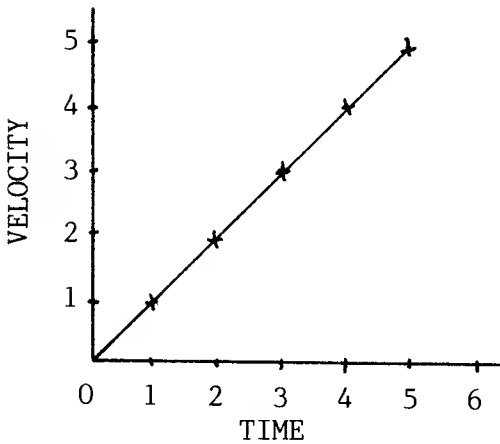
Therefore, the object remains stationary in the Y direction.

If a force were suddenly applied to our moving object so that the velocity in the X direction were to increase by one with each time frame, the distances traveled would grow substantially.

TIME	VELOCITY	POSITION (distance)
0	0	0
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21

$$VX = VX + 1$$

$$X = X + VX$$

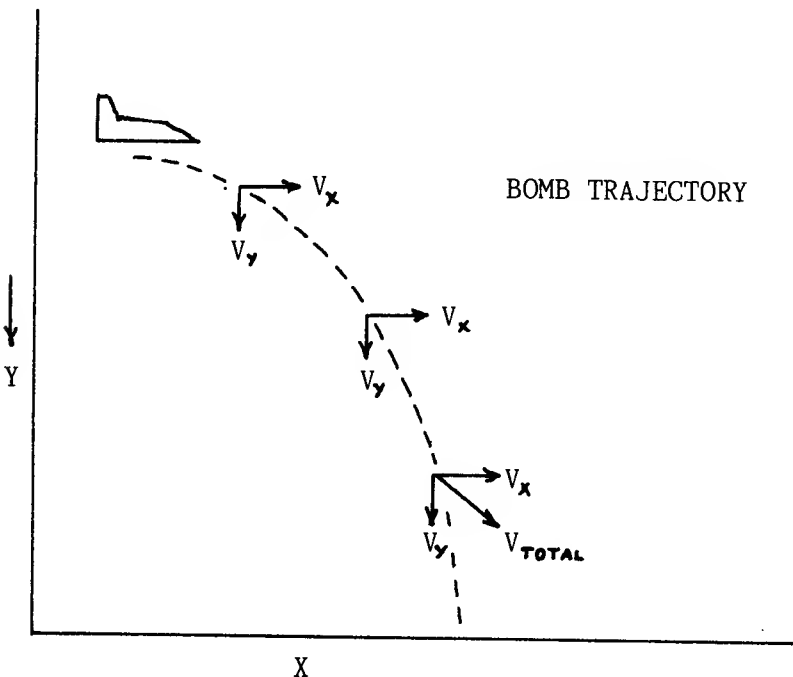


This driving force that speeds up our object is called acceleration ($V = V + A$). The acceleration in the previous example was $+1$ units/frame. The acceleration in space games is a rocket's thrust and, for falling bombs, it is gravity. To simplify things, when working with a falling bomb, we will neglect variables like wind resistance, and assume that the bomb has a small forward velocity equal to that of the plane. The plot of the trajectory of a falling bomb is shown below. The trajectory forms a curve that is often called "parabolic". You should note that although the velocity in the X direction remains constant, the velocity in the Y direction (V_Y) grows larger with time. It grows larger because gravity accelerates the object constantly in the downward direction. This same effect can be observed by dropping a ball from the second or third story of a building. At first, the ball falls slowly, but then it begins falling faster. Observers at ground level will note an accelerated moving ball just before it bounces.

The velocity of the falling bomb has two components represented by velocity vectors - one in the X direction and the other in the Y direction. These two velocity vectors can be graphically added together to form a total velocity vector. The summation of the two vectors determines the resultant direction of an object's motion for each animation frame. Since the V_Y vector grows larger with each frame, the total velocity vector begins to point downward. Eventually, the bomb will be falling almost straight down. Thus:

$$V_X = \text{CONST}$$

$$V_Y = V_Y + \text{GRAVITY}$$

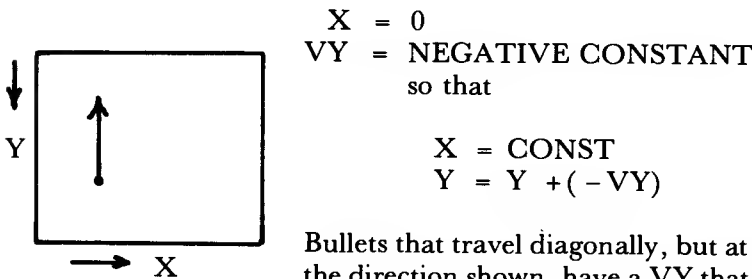


If you are programming the motion of a falling bomb, the equations or algorithm are as follows.

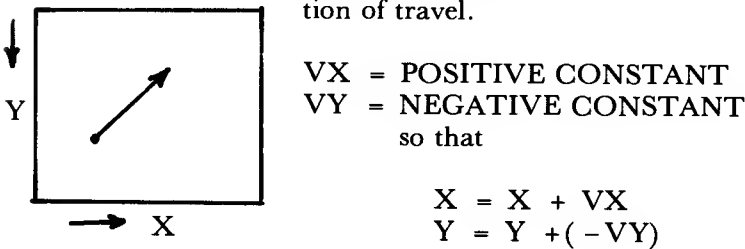
$$\begin{array}{lcl} \text{VX} & = & \text{CONST} \\ \text{VY} & = & \text{VY} + \text{GRAVITY} \end{array} \qquad \begin{array}{lcl} \text{X} & = & \text{X} + \text{VX} \\ \text{Y} & = & \text{Y} + \text{VY} \end{array}$$

For all practical purposes, a gravity constant of 3 to 5 will produce realistic curves on the Apple's Hi-Res screen, but this, again, like our choice of a constant for paddle movement, is dependent on factors like the animation frame rate and the scale of other objects on the screen.

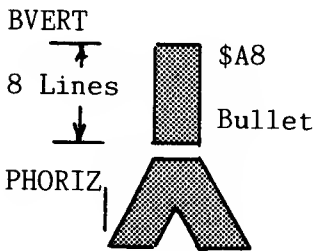
The trajectories of bullets and artillery shells are another useful feature in games. Bullets in games like Apple Invaders and Galaxian travel straight upwards on the screen.



Bullets that travel diagonally, but at a constant velocity in the direction shown, have a VY that is negative and a VX that is positive. The velocity vector determines the direction of travel.



Our bullet is fired from a movable gun base at the bottom of the screen. Its location, in relation to the gun barrel, is shown in the design at the right. The bullet's shape is eight units tall by four units wide and, like the gun base, uses seven different offset shape tables. Although the bullet is white, it is easier to use the same drawing routine to move it in conjunction with the gun base.

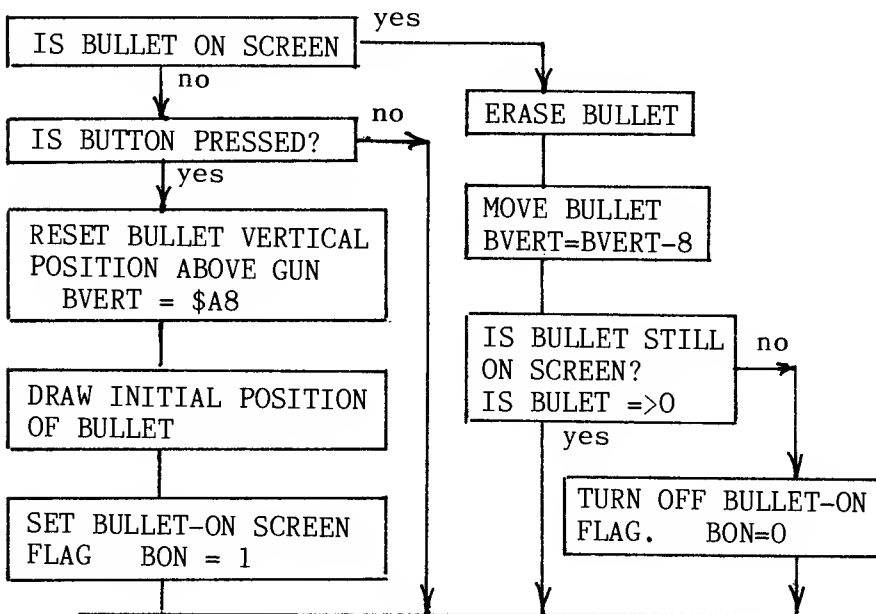


The bullet's horizontal velocity is $VX = 0$ and its vertical velocity is $VY = -8$. Thus, $X = X + VX$, or $X = \text{const}$, and $Y = Y - VY$. The bullet's vertical position is defined as $BVERT$. Therefore, $BVERT = BVERT - 8$ for each frame. If the bullet's horizontal position is to remain constant once it is fired, it must be set free of $PHORIZ$ (the gun's horizontal position), because its value would undoubtedly change if the gun turret moves after the bullet is fired. The bullet's horizontal position, $BPHORIZ$, is set equal to $PHORIZ$ when the gun fires, and is used to determine the horizontal offset into the screen line while it plots the bullet. The value is also used to index into the $XOFF$ table, which in turn acts as an index to the proper shape table when the bullet is plotted on the screen.

The bullet travels further toward the top of the screen during each screen frame. Notice that it travels exactly eight lines upwards per cycle. This allows us to begin drawing at the start of one of the 24 eight line subgroups.

The code also prevents you from firing more than one bullet at a time. When a bullet is on the screen, a flag called BON (short for "bullet on") is set to prevent you from firing again. There is more than a casual reason for doing this. If more than one bullet were fired at one time, you would need to keep track of each bullet's position separately. While two bullets might be manageable, a large number would involve storing the position values into tables, then accessing them in sequence during the bullet setup routine.

A flow chart of the algorithm and the code is shown below.

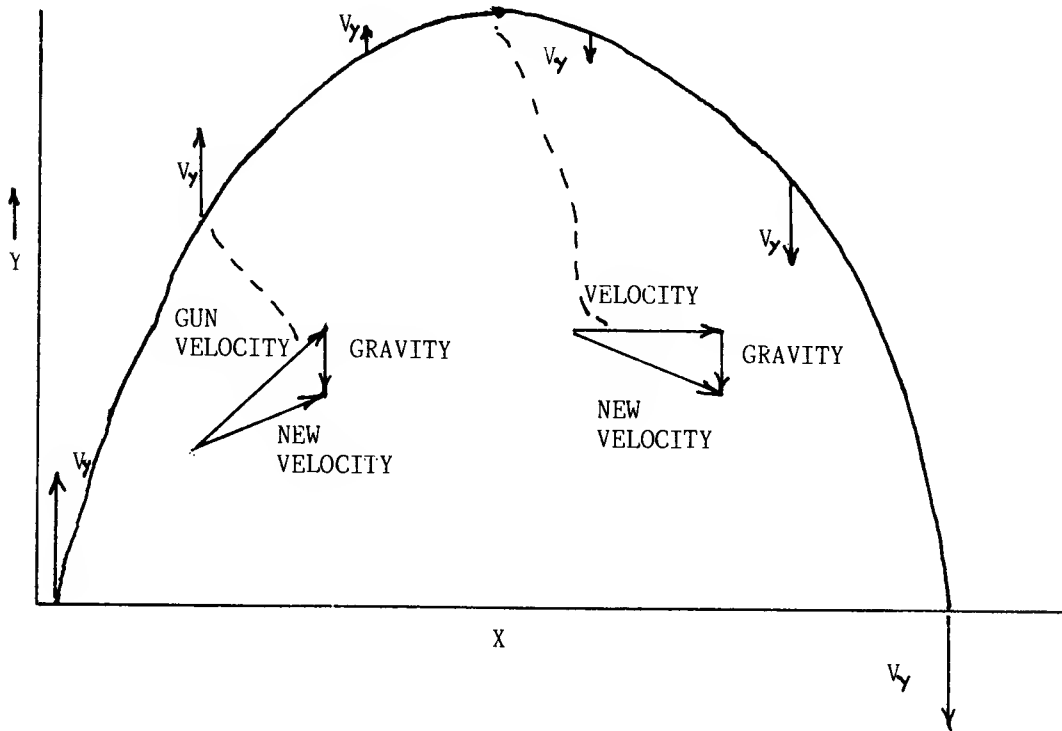


```

195 *BULLET SETUP
616D: AD 0D 60 196 BSETUP LDA BHORIZ
6170: 8D 0F 60 197 STA HORIZ
6173: AC 0E 60 198 LDY BPHORIZ
6176: BE 7C 64 199 LDX XOFF,Y ;INDEX TO WHICH SHAPE TABLE
6179: BD A2 65 200 LDA BSHPLO,X ;INDEX TO GET LO BYTE OF BOMB -
201 *- ;SHAPE TABLE
617C: 85 50 202 STA SHPL
617E: A9 67 203 LDA #>BSHAPES ;GET HI BYTE OF SHAPE
6180: 85 51 204 STA SHPH
6182: A9 02 205 LDA #$02
6184: 8D 13 60 206 STA SLNGH
6187: 8D 08 60 207 STA TEMP
618A: A9 07 208 LDA #$07 ;SHAPE 7 LINES DEEP
618C: 8D 12 60 209 STA DEPTH
618F: AD 15 60 210 LDA BVERT
6192: 8D 0A 60 211 STA TVERT
6195: 60 212 RTS
213 *BULLET SUBROUTINE
6196: AD 16 60 214 BULLET LDA BON ;TEST BULLET ON SCREEN
6199: C9 01 215 CMP #$01
619B: B0 27 216 BGE BULUPD
619D: AD 62 C0 217 LDA $CO62 ; NEG BUTTON PRESSED
61A0: 30 03 218 BMI FIRE1
61A2: 4C E3 61 219 JMP NOSHOOT
61A5: A9 A8 220 FIRE1 LDA #$A8
61A7: 8D 15 60 221 STA BVERT
61AA: AC 0B 60 222 LDY PHORIZ
61AD: 8C 0E 60 223 STY BPHORIZ ;BULLET HORIZ POS CONSTANT AT -
224 *- ;INITIAL FIRING POSITION(0-255)
61B0: B9 64 63 225 LDA XBASE,Y ;FIND HOR BYTE OFFSET
61B3: 8D 0D 60 226 STA BHORIZ ;(CONSTANT DURING VERTICAL TRAVEL)
61B6: 20 6D 61 227 JSR BSETUP
61B9: 20 A8 60 228 JSR GDRAW
61BC: A9 01 229 LDA #$01
61BE: 8D 16 60 230 STA BON ;SET BULLET ON SCREEN FLAG
61C1: 4C E3 61 231 JMP NOSHOOT
61C4: 20 6D 61 232 BULUPD JSR BSETUP
61C7: 20 A8 60 233 JSR GDRAW
61CA: 38 234 SEC
61CB: AD 15 60 235 LDA BVERT
61CE: E9 08 236 SBC #$08
61D0: 8D 15 60 237 STA BVERT ;THE CARRY FLAG IS SET IF POS
61D3: B0 08 238 BCS SKIP
61D5: A9 00 239 LDA #$00 ;SET BULLET DEAD FLAG
61D7: 8D 16 60 240 STA BON
61DA: 4C E3 61 241 JMP NOSHOOT
61DD: 20 6D 61 242 SKIP JSR BSETUP
61E0: 20 A8 60 243 JSR GDRAW
61E3: 60 244 NOSHOOT RTS

```


If you consider a bullet that is traveling diagonally upwards and to the right, and allow gravity to take effect, then the trajectory resembles that of an artillery shell.



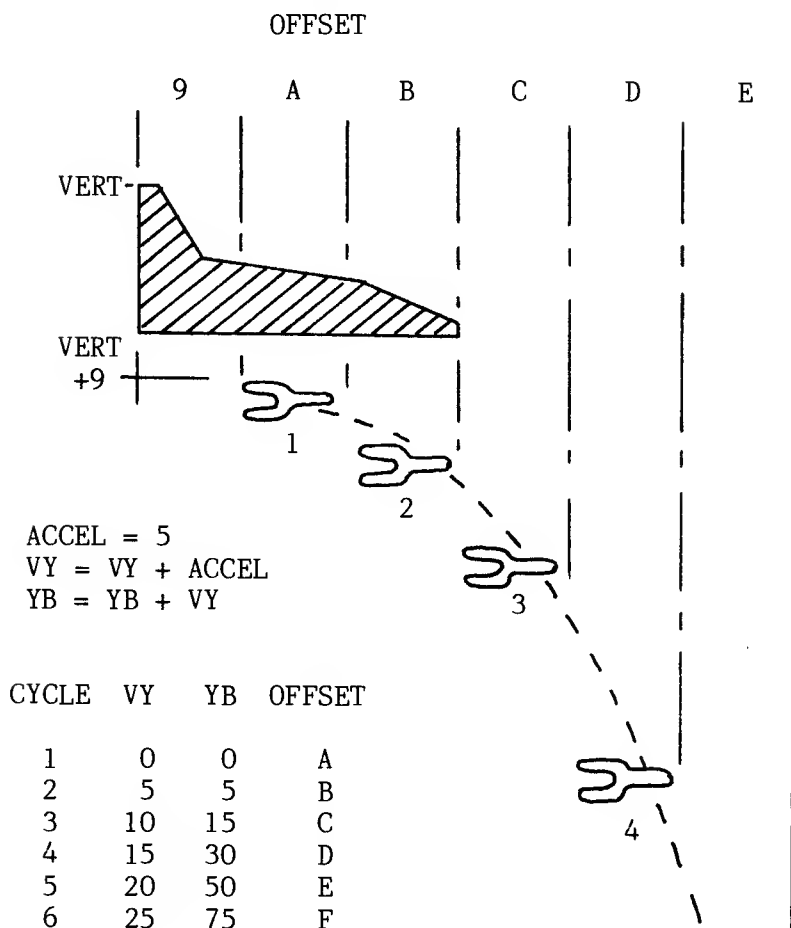
The gravity vector tends to bend our velocity vector so that it no longer travels at its initial 45 degree angle. By the time our bullet reaches the peak of its flight, the gravity vector has incrementally subtracted our vertical velocity vector to zero. At that point, there is only the horizontal velocity component. Since gravity affects our bullet at every time increment, it soon causes our velocity vector to have a negative vertical component. The bullet then begins to fall.

$$\begin{array}{ll} VY = VY + (-G) & Y = Y + VY \\ VX = \text{CONST} & X = X + VX \end{array}$$

Once you understand the vector concept of how an object falls, the bomb drop routine becomes elementary. The bomb must fall from the center of our plane because, by design, bomb bays are located at the plane's center of gravity. Since the tail of our plane is the vertical paddle position (VERT) and the plane is eight lines deep, the first available plotting position beneath the plane is at (VERT + 9).

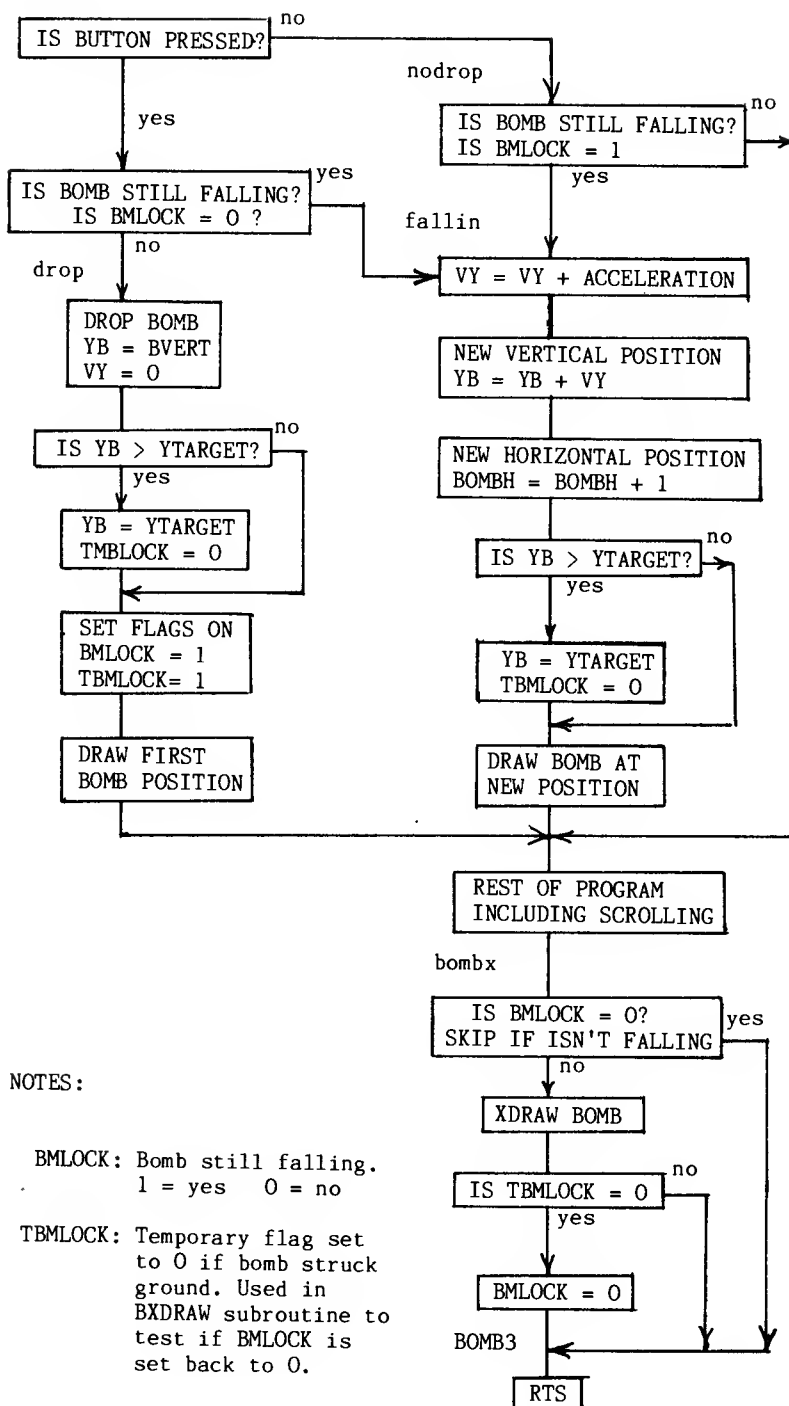
The bomb can be defined by the following shape table.

	●	●	●				07
		●	●	●	●	●	7E
	●	●	●				07



To simplify the graphics, it is easier to move the bomb horizontally one byte (or seven pixels) at a time. Consequently, with the bomb plotted in white, the even - odd offset color problems vanish. The flowchart and code follow.

bomb



NOTES:

BMLOCK: Bomb still falling.
1 = yes 0 = no

TBMLOCK: Temporary flag set
to 0 if bomb struck
ground. Used in
BXDRAW subroutine to
test if BMLOCK is
set back to 0.

```

607 *BOMB SUBROUTINE
608 *
6489: AD 61 C0 609 BOMB LDA $C061 ;NEG IF BUTTON PRESSED
648C: 30 03 610 BMI BOMB1
648E: 4C BD 64 611 JMP NODROP
6491: AD 1A 60 612 BOMB1 LDA BMLOCK
6494: C9 01 613 CMP #$01 ;IS BOMB STILL FALLING?
6496: B0 2A 614 BGE FALLIN ;YES, GOTO FALLIN
6498: AD 0C 60 615 DROP LDA VERT
649B: 18 616 CLC
649C: 69 09 617 ADC #$09
649E: 8D 16 60 618 STA BVERT ;INITIAL POSITION OF BOMB
64A1: 8D 17 60 619 STA TBVERT
64A4: A9 0A 620 LDA #$0A ;STARTING HORIZ POSITION
64A6: 8D 19 60 621 STA BHORIZ
64A9: A9 00 622 LDA #$00 ;INITIAL VERTICAL VELOCITY
64AB: 8D 18 60 623 STA BVELY
64AE: A9 01 624 LDA #$01
64B0: 8D 1A 60 625 STA BMLOCK ;RESET TO ON
64B3: 8D 1B 60 626 STA TBMLOCK ;RESET END OF FALL TO OFF
64B6: 20 45 64 627 JSR BSET
64B9: 20 59 64 628 JSR BDRAW ;DRAW BOMB
64BC: 60 629 RTS
64BD: AD 1A 60 630 NODROP LDA BMLOCK
64C0: F0 34 631 BEQ BOMB3 ;IS BOMB STILL FALLING
64C2: AD 18 60 632 FALLIN LDA BVELY
64C5: 18 633 CLC
64C6: 69 05 634 ADC #$05 ;ADD ACCELERATION CONSTANT
64C8: 8D 18 60 635 STA BVELY ;NEW VERTICAL VELOCITY
64CB: 6D 16 60 636 ADC BVERT
64CE: 8D 17 60 637 STA TBVERT
64D1: 8D 16 60 638 STA BVERT ;BOMB'S NEW VERTICAL POSITION
64D4: AD 19 60 639 LDA BHORIZ
64D7: 69 01 640 ADC #$01 ;BOMB'S HORIZ. VELOCITY(CONSTANT)
64D9: 8D 19 60 641 STA BHORIZ ;BOMB'S NEW HORIZ. POSITION
642 *TEMP DETECT FOR BOMB LANDING
64DC: AD 16 60 643 LDA BVERT
64DF: C9 B0 644 CMP #$B0 ;BOTTOM SCREEN?
64E1: 90 0D 645 BLT BOMB2 ;NO! THEN BOMB2
64E3: A9 B0 646 LDA #$B0
64E5: 8D 16 60 647 STA BVERT
64E8: 8D 17 60 648 STA TBVERT
64EB: A9 00 649 LDA #$00
64ED: 8D 1B 60 650 STA TBMLOCK ;SET END OF BOMB FALL FLAG
64F0: 20 45 64 651 BOMB2 JSR BSET
64F3: 20 59 64 652 JSR BDRAW
64F6: 60 653 BOMB3 RTS
654 *BOMB XDRAW
64F7: AD 1A 60 655 BOMBX LDA BMLOCK ;IS BOMB STILL FALLING?(1=YES)
64FA: F0 16 656 BEQ BOMBX1 ;SKIP IF 0
64FC: 20 45 64 657 JSR BSET
64FF: AD 16 60 658 LDA BVERT
6502: 8D 17 60 659 STA TBVERT
6505: 20 70 64 660 JSR BXDRAW ;XDRAW BOMB
6508: AD 1B 60 661 LDA TBMLOCK
650B: D0 05 662 BNE BOMBX1
650D: A9 00 663 LDA #$00
650F: 8D 1A 60 664 STA BMLOCK ;RESET BOMB FALLING TO OFF
6512: 60 665 BOMBX1 RTS

```

```

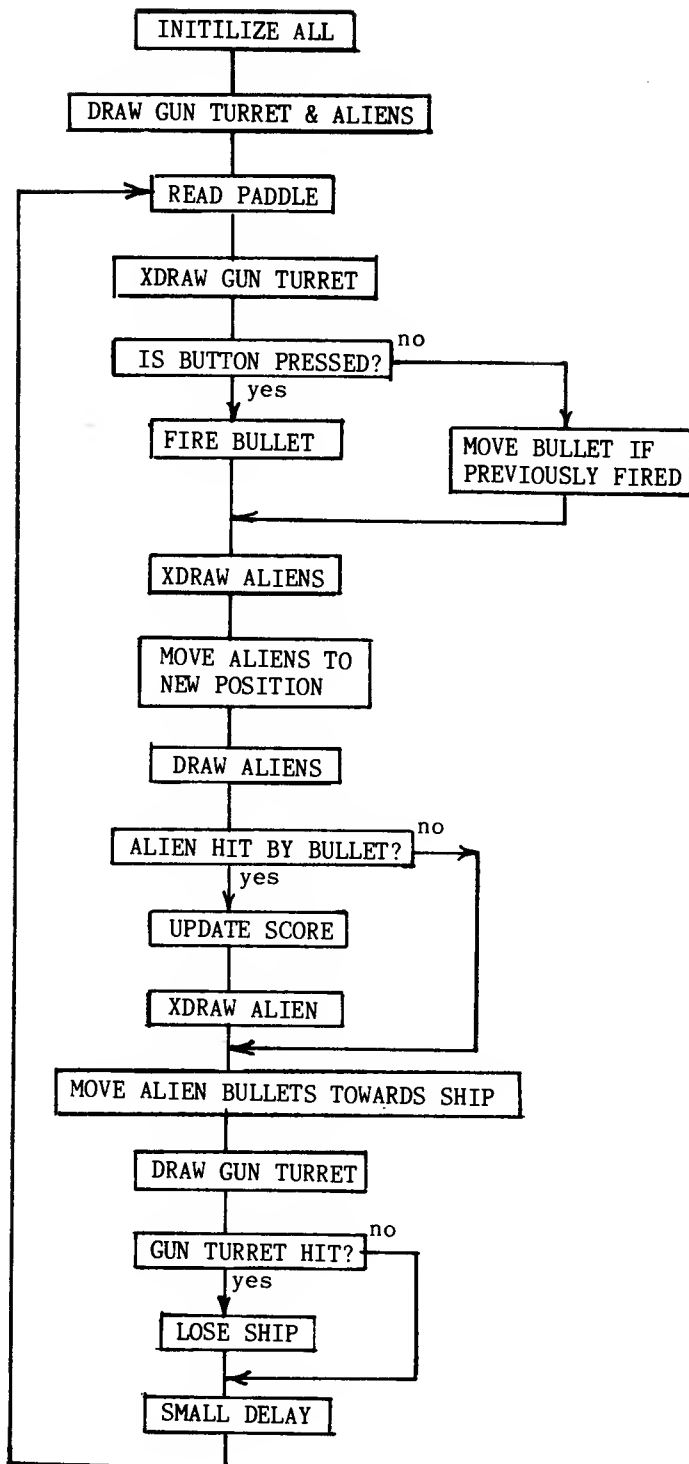
574 *DRAWING ROUTINES FOR BOMB
575 *
6445: A9 EF 576 BSET      LDA #<SHBOMB ;ADDRESS BOMB SHAPE
6447: 85 56 577      STA BOMBL
6449: A9 68 578      LDA #>SHBOMB
644B: 85 57 579      STA BOMBH
644D: AD 19 60 580     LDA BHORIZ ;BOMB'S HORIZ. POSITION
6450: 8D 0E 60 581     STA HORIZ
6453: A9 03 582      LDA #$03
6455: 8D 11 60 583     STA DEPTH
6458: 60 584      RTS
6459: AC 17 60 585     BDRAW    LDY TBVERT ;BOMB VERT POS
645C: 20 1C 63 586     JSR GETADR
645F: A2 00 587      LDX #$00
6461: A1 56 588      LDA (BOMBL,X) ;GET ADDRESS OF BOMB SHAPE
6463: 91 26 589      STA (HIRESL),Y ;PLOT
6465: EE 17 60 590     INC TBVERT
6468: E6 56 591      INC BOMBL
646A: CE 11 60 592     DEC DEPTH
646D: D0 EA 593      BNE BDRAW
646F: 60 594      RTS
6470: AC 17 60 595     BXDRAW   LDY TBVERT
6473: 20 1C 63 596     JSR GETADR
6476: A2 00 597      LDX #$00
6478: A1 56 598      LDA (BOMBL,X)
647A: 51 26 599      EOR (HIRESL),Y
647C: 91 26 600      STA (HIRESL),Y
647E: EE 17 60 601     INC TBVERT
6481: E6 56 602      INC BOMBL
6483: CE 11 60 603     DEC DEPTH
6486: D0 E8 604      BNE BXDRAW
6488: 60 605      RTS

```

THE INVADERS TYPE GAME

Games of this type are classed as shoot-'em-up games. They generally involve a movable gun turret, or space ship, that traverses the bottom of the screen. The object is to defend against a horde of attacking aliens by firing bullets up at them. The aliens can either advance in ranks, like they do in Space Invaders, or they can swoop down singly or in groups, as they do in Apple Galaxian. Sometimes, background stars, moving from top to bottom, generate the feeling that your gun or ship is in motion. But these games still involve a static screen in the sense that all objects are manipulated within the screen space.

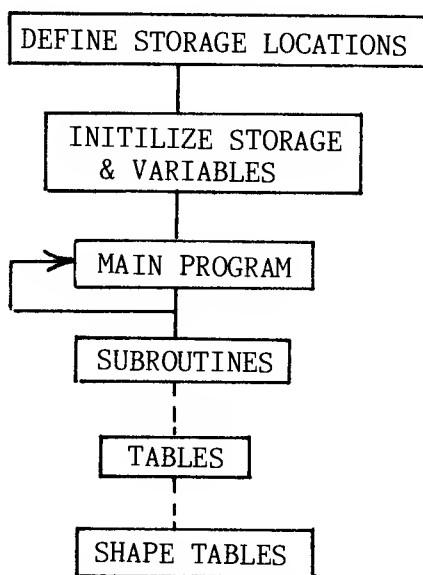
On the other hand, there are games that could be classed as dynamic because the entire background is scrolling in some preset direction, while the ship or other vehicle usually has controllable movement on the non-scrolling axis only. Objects which are out of view can be manipulated and scheduled to appear when your ship moves into their general vicinity. Moving your ship involves scrolling the entire background, so that terrain and objects out of the range of your display, suddenly appear. Of course, the terrain you previously



occupied is now off screen. Arcade games like Pegasus II involve constant terrain scrolling from right to left as your spaceship moves further into the enemy's territory. This type of animation will be discussed in the following chapter.

The sequence of events in an Invaders game is diagrammed above. It is typical of most games. While we aren't going to develop the entire game, we will integrate the paddle and bullet firing routines previously outlined in this chapter with the color drawing routines discussed in Chapter 5.

Since this is the first time that we have actually put together developed subroutines into a workable game, I should discuss the overall structure of a machine language program. Programs begin with storage allocations for variables, and zero page equates or assignments to specific memory locations in zero page for others. These are followed by initialization routines that activate Hi-Res graphics, clear the screen, and set specific variables to their initial values. The main program loop comes next, followed by subroutines. Your tables, both shape and reference, reside at the end.



Using a good assembler makes the job of writing a program relatively easy. All the tedious mechanical problems like relative addressing for branch instructions, references to variable storage, and memory storage assignments are handled automatically. In fact, the assembler is so adept at calculating addresses that I often use it for generating internal reference tables to the locations of my shapes.

Normally, it is good programming practice to put shape tables in some specific yet safe place in memory. But while developing short programs, it is an extra step to load your shape tables into memory each time that you want to test the program. Sometimes, it is more convenient to incorporate shape tables into your program, although their memory location changes with each modification to your source code.

The assembler can be used to define a reference table to the low byte of each shape in your shape table. In the TED II + assembler, DB defines a byte - the lo byte. BIG MAC and MERLIN use DFB.

```

659B: 16      SHPLO DB  SHAPES
659C: 2E              DB  SHAPES + $18
659D: 46              DB  SHAPES + $30

                DB  SHAPES + $90

```

The assembler looks up the lo byte address for each of our shapes according to the address that we give to it. Each shape is 24 (or \$18) bytes long. This accounts for the reason each succeeding shape address increases by \$18. Notice on the left of the above listing that the actual byte value is placed into our table for each shape.(SHPLO 16 2E 46 5E ...). This corresponds exactly to the lo byte values in our floating shape table. I'll extend a word of caution about using this method. Shape tables must not cross page boundaries, because the hi byte, which is stored at SHPH in our drawing routine, must be kept constant. Sometimes, extra space needs to be allocated in the code just before the shape table for correcting this problem. The DS pseudo-op code to Define Storage can be used.

The lo and hi bytes for a particular shape are determined by the following code:

```

LDY PHORIZ      ;PADDLE VALUE 0-255
LDX XOFF,Y      ;INDEX TO FIND WHICH SHAPE IN TABLE
LDA SHPLO,X     ;INDEX TO GET LO BYTE OF SHAPE IN TABLE
STA SHPL
LDA #>SHAPES    ;GET HI BYTE OF SHAPE TABLE
STA SHPH

```

If you were to choose, instead, to put the shape table at \$7000 in memory, you would use a table called SHPADR to index to the proper shape. Each position in the table would reference the lo byte of a shape in the shape table.

```

SHPADR  HEX  00 18 30 48 60 78 90

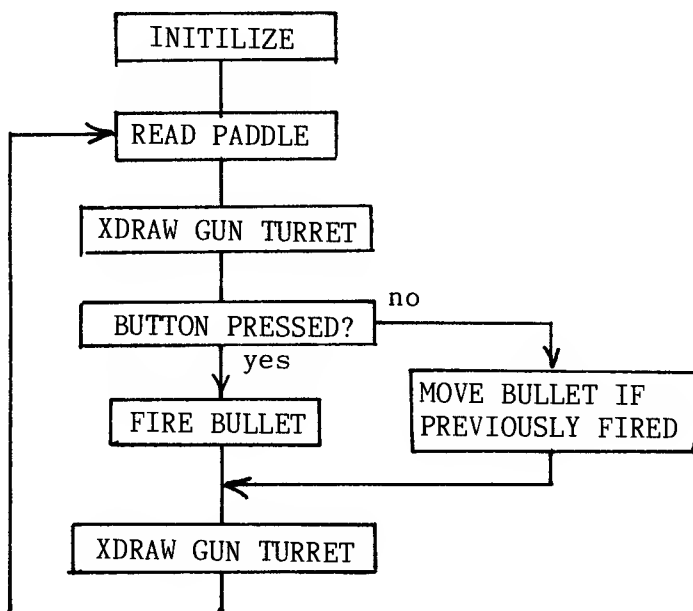
```


The setup routine is modified as follows:

```
LDY PHORIZ    ;PADDLE VALUE 0-256
LDX XOFF,Y     ;INDEX TO FIND WHICH SHAPE IN TABLE
LDA SHPADR,X   ;INDEX TO LO BYTE IN TABLE
STA SHPL
LDA $70        ;HI BYTE OF TABLE
STA SHPH
```

There are no speed advantages or disadvantages gained by using either method. The former method is strictly for convenience to be used while developing small programs. To avoid mistakes, large programs should definitely have shape tables fixed in memory.

The Invaders routine which follows lacks alien targets. It does, however, have a paddle-controlled gun turret which is capable of firing one bullet at a time. It is a start, and as you will see later, putting aliens on the screen is not difficult. A simple flow chart of the program and the actual code is shown below.



```

1      *CODE FOR PART OF INVADERS GAME
2      ORG  $6000
6000: 4C 17 60 3      JMP  PROG          ;JUMP TO START OF CODE
4      COUNT  DS  1
5      INDEX  DS  1
6      PADDLEL DS  1
7      PADDLEH DS  1
8      PDL     DS  1
9      TEMP    DS  1
10     VERT     DS  1
11     TVERT    DS  1
12     PHORIZ   DS  1
13     TPHORIZ  DS  1
14     BHORIZ   DS  1
15     BPHORIZ  DS  1
16     HORIZ    DS  1
17     OBJ      DS  1
18     LNGH     DS  1
19     DEPTH    DS  1
20     SLNGH    DS  1
21     SHOT     DS  1
22     BVERT    DS  1
23     BON      DS  1
24     HIRESL   EQU  $26
25     HIRESH   EQU  HIRESL+$1
26     SHPL     EQU  $50
27     SHPH     EQU  SHPL+$1
28     SSHPL    EQU  $52
29     SSHPH    EQU  $53
30     STESTL   EQU  $54
31     STESTH   EQU  STESTL+$1
32     PREAD    EQU  $FB1E
6017: AD 50 CO 33     PROG      LDA  $C050
601A: AD 52 CO 34     LDA  $C052
601D: AD 57 CO 35     LDA  $C057
6020: 20 8E 60 36     JSR  CLRSCR
6023: A9 00 37        LDA  #$00
6025: 8D 16 60 38     STA  BON
39     *READ PADDLE #1
6028: A2 01 40     RPD L     LDX  #$01
602A: 20 1E FB 41     JSR  PREAD
602D: 8C 07 60 42     SKIPP   STY  PDL
43     TYA
6031: CD 0B 60 44     CMP  PHORIZ      ;PADDLE<HORIZ POS THEN SUBTRACT 5
6034: B0 1E 45     BGE  PADDLE3
6036: AD 0B 60 46     LDA  PHORIZ
6039: 38 47     SEC
603A: E9 05 48     SBC  #$05
603C: B0 08 49     BGE  PADDLE1      ;MAKE SURE =>0
603E: A9 00 50     LDA  #$00
6040: 8D 0B 60 51     STA  PHORIZ
6043: 8D 0C 60 52     STA  TPHORIZ
6046: CD 07 60 53     PADDLE1 CMP  PDL      ;DON'T WANT TO GO PAST PADDLE POS
6049: B0 03 54     BGE  PADDLE2
604B: AD 07 60 55     LDA  PDL
604E: 8D 0B 60 56     PADDLE2 STA  PHORIZ
6051: 4C 71 60 57     JMP  PADDLE6
6054: CD 0B 60 58     PADDLE3 CMP  PHORIZ      ;PADDLE>PHORIZ POS THEN ADD 5
6057: F0 12 59     BEQ  PADDLE4
6059: AD 0B 60 60     LDA  PHORIZ

```

```

605C: C9 FA 61      CMP  #$FA      ;IS PHORIZ>250
605E: B0 OB 62      BGE  PADDLE4
6060: AD OB 60 63    LDA  PHORIZ
6063: 18 64          CLC
6064: 69 05 65      ADC  #$05
6066: CD 07 60 66    CMP  PDL      ;DON'T WANT TO GO PAST PADDLE POS
6069: 90 03 67      BLT  PADDLE5
606B: AD 07 60 68    PADDLE4 LDA  PDL
606E: 8D OB 60 69    PADDLE5 STA  PHORIZ
6071: 8D OC 60 70    PADDLE6 STA  TPHORIZ
6074: 20 3F 61 71    JSR  GSETUP
6077: 20 A8 60 72    JSR  GDRAW
607A: 20 6D 61 73    JSR  BSETUP
607D: 20 96 61 74    JSR  BULLET
6080: A9 60 75      LDA  #$60
6082: 20 A8 FC 76    JSR  $FCA8
6085: 20 3F 61 77    JSR  GSETUP
6088: 20 A8 60 78    JSR  GDRAW
608B: 4C 28 60 79    JMP  RPDL      ;BACK TO BEGINNING OF MAIN LOOP
      80      *
      81      ** S U B R O U T I N E S **
      82      *
      83      *CLEAR SCREEN
608E: A9 00 84      CLRSCR LDA  #$00
6090: 85 26 85      STA  HIRESL
6092: A9 20 86      LDA  #$20
6094: 85 27 87      STA  HIRESH
6096: A0 00 88      CLR1  LDY  #$00
6098: A9 00 89      LDA  #$00
609A: 91 26 90      CLR2  STA  (HIRESL),Y
609C: C8 91          INY
609D: D0 FB 92      BNE  CLR2
609F: E6 27 93      INC  HIRESH
60A1: A5 27 94      LDA  HIRESH
60A3: C9 40 95      CMP  #$40
60A5: 90 EF 96      BCC  CLR1
60A7: 60 97          RTS
      98      *DRAW GUN SHAPE DEPTH LINES BY LNGH
60A8: AC 0A 60 99    GDRAW LDY  TVERT      ; VERTICAL POSITION
60AB: 20 E6 60 100   JSR  GETADR
60AE: A2 00 101      LDX  #$00
60B0: A1 50 102      GDRAW3 LDA  (SHPL,X) ;GET BYTE OF SHIP'S SHAPE
60B2: 51 26 103      EOR  (HIRESL),Y
60B4: 91 26 104      STA  (HIRESL),Y ;PLOT
60B6: E6 50 105      INC  SHPL      ; NEXT BYTE OF TABLE
60B8: C8 106          INY
60B9: CE 13 60 107   DEC  SLNGH
60BC: D0 F2 108      BNE  GDRAW3 ;IF LINE NOT FINISHED BRANCH
60BE: EE 0A 60 109   INC  TVERT ;OTHERWISE NEXT LINE DOWN
60C1: CE 12 60 110   DEC  DEPTH
60C4: D0 E2 111      BNE  GDRAW
60C6: 60 112          RTS
      113      *XDRAW GUN SHAPE
60C7: AC 0A 60 114    GXDRAW LDY  TVERT      ;VERTICAL POSITION
60CA: 20 E6 60 115   JSR  GETADR
60CD: A2 00 116      LDX  #$00
60CF: A1 50 117      GXDRAW2 LDA  (SHPL,X)
60D1: 51 26 118      EOR  (HIRESL),Y
60D3: 91 26 119      STA  (HIRESL),Y
60D5: E6 50 120      INC  SHPL

```

60D7:	C8	121	INY	
60D8:	CE 13 60	122	DEC	SLNGH
60DB:	DO F2	123	BNE	GXDRAW2
60DD:	EE 0A 60	124	INC	TVERT
60EO:	CE 12 60	125	DEC	DEPTH
60E3:	DO E2	126	BNE	GXDRAW
60E5:	60	127	RTS	
		128	*GETADR	SUBROUTINE
60E6:	B9 E4 61	129	GETADR	LDA YVERTL,Y ;LOOK UP LO BYTE OF LINE
60E9:	18	130	CLC	
60EA:	6D 0F 60	131	ADC	HORIZ ;ADD DISPLACEMENT INTO LINE
60ED:	85 26	132	STA	HIRESL
60EF:	B9 A4 62	133	LDA	YVERTH,Y ;LOOK UP HI BYTE OF LINE
60F2:	85 27	134	STA	HIRESH
60F4:	AD 08 60	135	LDA	TEMP
60F7:	8D 13 60	136	STA	SLNGH
60FA:	A0 00	137	LDY	#\$00
60FC:	60	138	RTS	
		139	*DRAW	ALIEN SHIPS & TARGETS
60FD:	A2 00	140	DRAW	LDX #\$00
60FF:	A1 50	141	DRAW2	LDA (SHPL,X)
6101:	91 26	142		STA (HIRESL),Y
6103:	A5 27	143	LDA	HIRESH
6105:	18	144	CLC	
6106:	69 04	145	ADC	#\$04
6108:	85 27	146	STA	HIRESH
610A:	E6 50	147	INC	SHPL
610C:	C9 40	148	CMP	#\$40
610E:	90 EF	149	BCC	DRAW2
6110:	E9 20	150	SBC	#\$20
6112:	85 27	151	STA	HIRESH
6114:	CE 11 60	152	DEC	LNHG
6117:	FO 03	153	BEQ	DRAW3
6119:	C8	154	INY	
611A:	DO E3	155	BNE	DRAW2
611C:	60	156	DRAW3	RTS
		157	*XDRAW	ALIEN SHIPS & TARGETS
611D:	A2 00	158	XDRAW	LDX #\$00
611F:	A1 50	159	XDRAW2	LDA (SHPL,X)
6121:	51 26	160		EOR (HIRESL),Y
6123:	91 26	161	STA	(HIRESL),Y
6125:	A5 27	162	LDA	HIRESH
6127:	18	163	CLC	
6128:	69 04	164	ADC	#\$04
612A:	85 27	165	STA	HIRESH
612C:	E6 50	166	INC	SHPL
612E:	C9 40	167	CMP	#\$40
6130:	90 ED	168	BCC	XDRAW2
6132:	E9 20	169	SBC	#\$20
6134:	85 27	170	STA	HIRESH
6136:	CE 11 60	171	DEC	LNHG
6139:	FO 03	172	BEQ	XDRAW3
613B:	C8	173	INY	
613C:	DO E1	174	BNE	XDRAW2
613E:	60	175	XDRAW3	RTS
		176	*DRAWING	ROUTINES SETUP
613F:	AC 0B 60	177	GSETUP	LDY PHORIZ ;PADDLE VALUE 0-256
6142:	B9 64 63	178		LDA XBASE,Y ;GET BYTE OFFSET IN TABLE
6145:	8D 0F 60	179	STA	HORIZ
6148:	BE 7C 64	180	LDX	XOFF,Y ;INDEX TO FIND WHICH SHAPE TABLE

614B: BC 94 65 181	LDY SHPADR,X	;X IS 0-6
614E: B9 9B 65 182	LDA SHPLO,Y	;INDEX TO GET LO BYTE SHAPE TABLE
6151: 85 50 183	STA SHPL	
6153: A9 66 184	LDA #>SHAPES	;GET HI BYTE OF SHAPE
6155: 85 51 185	STA SHPH	
6157: A9 03 186	LDA #\$03	
6159: 8D 13 60 187	STA SLNGH	
615C: 8D 08 60 188	STA TEMP	
615F: A9 08 189	LDA #\$08	
6161: 8D 12 60 190	STA DEPTH	
6164: A9 B0 191	LDA #\$B0	
6166: 8D 09 60 192	STA VERT	
6169: 8D 0A 60 193	STA TVERT	
616C: 60 194	RTS	
195	*BULLET SETUP	
616D: AD 0D 60 196	BSETUP LDA BHORIZ	
6170: 8D 0F 60 197	STA HORIZ	
6173: AC 0E 60 198	LDY BPHORIZ	
6176: BE 7C 64 199	LDX XOFF,Y	;INDEX TO WHICH SHAPE TABLE
6179: BD A2 65 200	LDA BSHPLO,X	;INDEX TO GET LO BYTE OF BOMB -
201	*-	;SHAPE TABLE
617C: 85 50 202	STA SHPL	
617E: A9 67 203	LDA #>BSHAPES	;GET HI BYTE OF SHAPE
6180: 85 51 204	STA SHPH	
6182: A9 02 205	LDA #\$02	
6184: 8D 13 60 206	STA SLNGH	
6187: 8D 08 60 207	STA TEMP	
618A: A9 07 208	LDA #\$07	;SHAPE 7 LINES DEEP
618C: 8D 12 60 209	STA DEPTH	
618F: AD 15 60 210	LDA BVERT	
6192: 8D 0A 60 211	STA TVERT	
6195: 60 212	RTS	
213	*BULLET SUBROUTINE	
6196: AD 16 60 214	BULLET LDA BON	;TEST BULLET ON SCREEN
6199: C9 01 215	CMP #\$01	
619B: B0 27 216	BGE BULUPD	
619D: AD 62 C0 217	LDA \$C062	; NEG BUTTON PRESSED
61A0: 30 03 218	BMI FIRE1	
61A2: 4C E3 61 219	JMP NOSHOOT	
61A5: A9 A8 220	FIRE1 LDA #\$A8	
61A7: 8D 15 60 221	STA BVERT	
61AA: AC 0B 60 222	LDY PHORIZ	
61AD: 8C 0E 60 223	STY BPHORIZ	;BULLET HORIZ POS CONSTANT AT -
224	*-	;INITIAL FIRING POSITION(0-255)
61B0: B9 64 63 225	LDA XBASE,Y	;FIND HOR BYTE OFFSET
61B3: 8D 0D 60 226	STA BHORIZ	; (CONSTANT DURING VERTICAL TRAVEL)
61B6: 20 6D 61 227	JSR BSETUP	
61B9: 20 A8 60 228	JSR GDRAW	
61BC: A9 01 229	LDA #\$01	
61BE: 8D 16 60 230	STA BON	;SET BULLET ON SCREEN FLAG
61C1: 4C E3 61 231	JMP NOSHOOT	
61C4: 20 6D 61 232	BULUPD JSR BSETUP	
61C7: 20 A8 60 233	JSR GDRAW	
61CA: 38 234	SEC	
61CB: AD 15 60 235	LDA BVERT	
61CE: E9 08 236	SBC #\$08	
61D0: 8D 15 60 237	STA BVERT	;THE CARRY FLAG IS SET IF POS
61D3: B0 08 238	BCS SKIP	
61D5: A9 00 239	LDA #\$00	;SET BULLET DEAD FLAG
61D7: 8D 16 60 240	STA BON	

```

61DA: 4C E3 61 241      JMP  NOSHOOT
61DD: 20 6D 61 242      SKIP JSR  BSETUP
61E0: 20 A8 60 243      JSR  GDRAW
61E3: 60          244      NOSHOOT RTS
          245      *
          246      **T A B L E S **
          247      *

61E4: 00 00 00
61E7: 00 00 00
61EA: 00 00 248      YVERTL  HEX  0000000000000000
61EC: 80 80 80
61EF: 80 80 80
61F2: 80 80 249      HEX  8080808080808080
61F4: 00 00 00
61F7: 00 00 00
61FA: 00 00 250      HEX  0000000000000000
61FC: 80 80 80
61FF: 80 80 80
6202: 80 80 251      HEX  8080808080808080
6204: 00 00 00
6207: 00 00 00
620A: 00 00 252      HEX  0000000000000000
620C: 80 80 80
620F: 80 80 80
6212: 80 80 253      HEX  8080808080808080
6214: 00 00 00
6217: 00 00 00
621A: 00 00 254      HEX  0000000000000000
621C: 80 80 80
621F: 80 80 80
6222: 80 80 255      HEX  8080808080808080
6224: 28 28 28
6227: 28 28 28
622A: 28 28 256      HEX  2828282828282828
622C: A8 A8 A8
622F: A8 A8 A8
6232: A8 A8 257      HEX  A8A8A8A8A8A8A8A8
6234: 28 28 28
6237: 28 28 28
623A: 28 28 258      HEX  2828282828282828
623C: A8 A8 A8
623F: A8 A8 A8
6242: A8 A8 259      HEX  A8A8A8A8A8A8A8A8
6244: 28 28 28
6247: 28 28 28
624A: 28 28 260      HEX  2828282828282828
624C: A8 A8 A8
624F: A8 A8 A8
6252: A8 A8 261      HEX  A8A8A8A8A8A8A8A8
6254: 28 28 28
6257: 28 28 28
625A: 28 28 262      HEX  2828282828282828
625C: A8 A8 A8
625F: A8 A8 A8
6262: A8 A8 263      HEX  A8A8A8A8A8A8A8A8
6264: 50 50 50
6267: 50 50 50
626A: 50 50 264      HEX  5050505050505050
626C: D0 D0 D0
626F: D0 D0 D0
6272: D0 D0 265      HEX  D0D0D0D0D0D0D0D0

```

6274:	50 50 50			
6277:	50 50 50			
627A:	50 50	266	HEX	5050505050505050
627C:	D0 D0 D0			
627F:	D0 D0 D0			
6282:	D0 D0	267	HEX	D0D0D0D0D0D0D0D0
6284:	50 50 50			
6287:	50 50 50			
628A:	50 50	268	HEX	5050505050505050
628C:	D0 D0 D0			
628F:	D0 D0 D0			
6292:	D0 D0	269	HEX	D0D0D0D0D0D0D0D0
6294:	50 50 50			
6297:	50 50 50			
629A:	50 50	270	HEX	5050505050505050
629C:	D0 D0 D0			
629F:	D0 D0 D0			
62A2:	D0 D0	271	HEX	D0D0D0D0D0D0D0D0
		272 *		
62A4:	20 24 28			
62A7:	2C 30 34			
62AA:	38 3C	273	YVERTH	HEX 2024282C3034383C
62AC:	20 24 28			
62AF:	2C 30 34			
62B2:	38 3C	274	HEX	2024282C3034383C
62B4:	21 25 29			
62B7:	2D 31 35			
62BA:	39 3D	275	HEX	2125292D3135393D
62BC:	21 25 29			
62BF:	2D 31 35			
62C2:	39 3D	276	HEX	2125292D3135393D
62C4:	22 26 2A			
62C7:	2E 32 36			
62CA:	3A 3E	277	HEX	22262A2E32363A3E
62CC:	22 26 2A			
62CF:	2E 32 36			
62D2:	3A 3E	278	HEX	22262A2E32363A3E
62D4:	23 27 2B			
62D7:	2F 33 37			
62DA:	3B 3F	279	HEX	23272B2F33373B3F
62DC:	23 27 2B			
62DF:	2F 33 37			
62E2:	3B 3F	280	HEX	23272B2F33373B3F
62E4:	20 24 28			
62E7:	2C 30 34			
62EA:	38 3C	281	HEX	2024282C3034383C
62EC:	20 24 28			
62EF:	2C 30 34			
62F2:	38 3C	282	HEX	2024282C3034383C
62F4:	21 25 29			
62F7:	2D 31 35			
62FA:	39 3D	283	HEX	2125292D3135393D
62FC:	21 25 29			
62FF:	2D 31 35			
6302:	39 3D	284	HEX	2125292D3135393D
6304:	22 26 2A			
6307:	2E 32 36			
630A:	3A 3E	285	HEX	22262A2E32363A3E
630C:	22 26 2A			
630F:	2E 32 36			

6312: 3A 3E	286	HEX	22262A2E32363A3E
6314: 23 27 2B			
6317: 2F 33 37			
631A: 3B 3F	287	HEX	23272B2F33373B3F
631C: 23 27 2B			
631F: 2F 33 37			
6322: 3B 3F	288	HEX	23272B2F33373B3F
6324: 20 24 28			
6327: 2C 30 34			
632A: 38 3C	289	HEX	2024282C3034383C
632C: 20 24 28			
632F: 2C 30 34			
6332: 38 3C	290	HEX	2024282C3034383C
6334: 21 25 29			
6337: 2D 31 35			
633A: 39 3D	291	HEX	2125292D3135393D
633C: 21 25 29			
633F: 2D 31 35			
6342: 39 3D	292	HEX	2125292D3135393D
6344: 22 26 2A			
6347: 2E 32 36			
634A: 3A 3E	293	HEX	22262A2E32363A3E
634C: 22 26 2A			
634F: 2E 32 36			
6352: 3A 3E	294	HEX	22262A2E32363A3E
6354: 23 27 2B			
6357: 2F 33 37			
635A: 3B 3F	295	HEX	23272B2F33373B3F
635C: 23 27 2B			
635F: 2F 33 37			
6362: 3B 3F	296	HEX	23272B2F33373B3F
6364: 00 00 00			
6367: 00 00 00			
636A: 00	297	XBASE	HEX 00000000000000
636B: 00 01 01			
636E: 01 01 01			
6371: 01	298	HEX	00010101010101
6372: 02 02 02			
6375: 02 02 02			
6378: 02	299	HEX	02020202020202
6379: 02 03 03			
637C: 03 03 03			
637F: 03	300	HEX	02030303030303
6380: 04 04 04			
6383: 04 04 04			
6386: 04	301	HEX	04040404040404
6387: 04 05 05			
638A: 05 05 05			
638D: 05	302	HEX	04050505050505
638E: 06 06 06			
6391: 06 06 06			
6394: 06	303	HEX	06060606060606
6395: 06 07 07			
6398: 07 07 07			
639B: 07	304	HEX	06070707070707
639C: 08 08 08			
639F: 08 08 08			
63A2: 08	305	HEX	08080808080808
63A3: 08 09 09			
63A6: 09 09 09			

63A9: 09	306	HEX 08090909090909
63AA: 0A 0A 0A		
63AD: 0A 0A 0A		
63B0: 0A	307	HEX 0A0A0A0A0A0A0A
63B1: 0A 0B 0B		
63B4: 0B 0B 0B		
63B7: 0B	308	HEX 0A0B0B0B0B0B0B
63B8: 0C 0C 0C		
63BB: 0C 0C 0C		
63BE: 0C	309	HEX 0C0C0C0C0C0C0C
63BF: 0C 0D 0D		
63C2: 0D 0D 0D		
63C5: 0D	310	HEX 0C0D0D0D0D0D0D
63C6: 0E 0E 0E		
63C9: 0E 0E 0E		
63CC: 0E	311	HEX 0E0E0E0E0E0E0E
63CD: 0E 0F 0F		
63D0: 0F 0F 0F		
63D3: 0F	312	HEX 0E0F0F0F0F0F0F
63D4: 10 10 10		
63D7: 10 10 10		
63DA: 10	313	HEX 10101010101010
63DB: 10 11 11		
63DE: 11 11 11		
63E1: 11	314	HEX 10111111111111
63E2: 12 12 12		
63E5: 12 12 12		
63E8: 12	315	HEX 12121212121212
63F9: 12 13 13		
63EC: 13 13 13		
63EF: 13	316	HEX 12131313131313
63F0: 14 14 14		
63F3: 14 14 14		
63F6: 14	317	HEX 14141414141414
63F7: 14 15 15		
63FA: 15 15 15		
63FD: 15	318	HEX 14151515151515
63FE: 16 16 16		
6401: 16 16 16		
6404: 16	319	HEX 16161616161616
6405: 16 17 17		
6408: 17 17 17		
640B: 17	320	HEX 16171717171717
640C: 18 18 18		
640F: 18 18 18		
6412: 18	321	HEX 18181818181818
6413: 18 19 19		
6416: 19 19 19		
6419: 19	322	HEX 18191919191919
641A: 1A 1A 1A		
641D: 1A 1A 1A		
6420: 1A	323	HEX 1A1A1A1A1A1A1A
6421: 1A 1B 1B		
6424: 1B 1B 1B		
6427: 1B	324	HEX 1A1B1B1B1B1B1B
6428: 1C 1C 1C		
642B: 1C 1C 1C		
642E: 1C	325	HEX 1C1C1C1C1C1C1C
642F: 1C 1D 1D		
6432: 1D 1D 1D		

6435: 1D	326	HEX	1C1D1D1D1D1D1D
6436: 1E 1E 1E			
6439: 1E 1E 1E			
643C: 1E	327	HEX	1E1E1E1E1E1E1E
643D: 1E 1F 1F			
6440: 1F 1F 1F			
6443: 1F	328	HEX	1E1F1F1F1F1F1F
6444: 20 20 20			
6447: 20 20 20			
644A: 20	329	HEX	20202020202020
644B: 20 21 21			
644E: 21 21 21			
6451: 21	330	HEX	20212121212121
6452: 22 22 22			
6455: 22 22 22			
6458: 22	331	HEX	22222222222222
6459: 22 23 23			
645C: 23 23 23			
645F: 23	332	HEX	22232323232323
6460: 24 24 24			
6463: 24 24 24			
6466: 24	333	HEX	24242424242424
6467: 24 25 25			
646A: 25 25 25			
646D: 25	334	HEX	24252525252525
646E: 26 26 26			
6471: 26 26 26			
6474: 26	335	HEX	26262626262626
6475: 26 27 27			
6478: 27 27 27			
647B: 27	336	HEX	26272727272727
647C: 00 00 01			
647F: 01 02 02			
6482: 03	337	XOFF	HEX 00000101020203
6483: 03 04 04			
6486: 05 05 06			
6489: 06	338	HEX	03040405050606
648A: 00 00 01			
648D: 01 02 02			
6490: 03	339	HEX	00000101020203
6491: 03 04 04			
6494: 05 05 06			
6497: 06	340	HEX	03040405050606
6498: 00 00 01			
649B: 01 02 02			
649E: 03	341	HEX	00000101020203
649F: 03 04 04			
64A2: 05 05 06			
64A5: 06	342	HEX	03040405050606
64A6: 00 00 01			
64A9: 01 02 02			
64AC: 03	343	HEX	00000101020203
64AD: 03 04 04			
64B0: 05 05 06			
64B3: 06	344	HEX	03040405050606
64B4: 00 00 01			
64B7: 01 02 02			
64BA: 03	345	HEX	00000101020203
64BB: 03 04 04			
64BE: 05 05 06			

64C1: 06	346	HEX 03040405050606
64C2: 00 00 01		
64C5: 01 02 02		
64C8: 03	347	HEX 00000101020203
64C9: 03 04 04		
64CC: 05 05 06		
64CF: 06	348	HEX 03040405050606
64D0: 00 00 01		
64D3: 01 02 02		
64D6: 03	349	HEX 00000101020203
64D7: 03 04 04		
64DA: 05 05 06		
64DD: 06	350	HEX 03040405050606
64DE: 00 00 01		
64E1: 01 02 02		
64E4: 03	351	HEX 00000101020203
64E5: 03 04 04		
64E8: 05 05 06		
64EB: 06	352	HEX 03040405050606
64EC: 00 00 01		
64EF: 01 02 02		
64F2: 03	353	HEX 00000101020203
64F3: 03 04 04		
64F6: 05 05 06		
64F9: 06	354	HEX 03040405050606
64FA: 00 00 01		
64FD: 01 02 02		
6500: 03	355	HEX 00000101020203
6501: 03 04 04		
6504: 05 05 06		
6507: 06	356	HEX 03040405050606
6508: 00 00 01		
650B: 01 02 02		
650E: 03	357	HEX 00000101020203
650F: 03 04 04		
6512: 05 05 06		
6515: 06	358	HEX 03040405050606
6516: 00 00 01		
6519: 01 02 02		
651C: 03	359	HEX 00000101020203
651D: 03 04 04		
6520: 05 05 06		
6523: 06	360	HEX 03040405050606
6524: 00 00 01		
6527: 01 02 02		
652A: 03	361	HEX 00000101020203
652B: 03 04 04		
652E: 05 05 06		
6531: 06	362	HEX 03040405050606
6532: 00 00 01		
6535: 01 02 02		
6538: 03	363	HEX 00000101020203
6539: 03 04 04		
653C: 05 05 06		
653F: 06	364	HEX 03040405050606
6540: 00 00 01		
6543: 01 02 02		
6546: 03	365	HEX 00000101020203
6547: 03 04 04		
654A: 05 05 06		

654D: 06	366	HEX	03040405050606
654E: 00 00 01			
6551: 01 02 02			
6554: 03	367	HEX	00000101020203
6555: 03 04 04			
6558: 05 05 06			
655B: 06	368	HEX	03040405050606
655C: 00 00 01			
655F: 01 02 02			
6562: 03	369	HEX	00000101020203
6563: 03 04 04			
6566: 05 05 06			
6569: 06	370	HEX	03040405050606
656A: 00 00 01			
656D: 01 02 02			
6570: 03	371	HEX	00000101020203
6571: 03 04 04			
6574: 05 05 06			
6577: 06	372	HEX	03040405050606
6578: 00 00 01			
657B: 01 02 02			
657E: 03	373	HEX	00000101020203
657F: 03 04 04			
6582: 05 05 06			
6585: 06	374	HEX	03040405050606
6586: 00 00 01			
6589: 01 02 02			
658C: 03	375	HEX	00000101020203
658D: 03 04 04			
6590: 05 05 06			
6593: 06	376	HEX	03040405050606
	377	*TABLES	
6594: 00 01 02			
6597: 03 04 05			
659A: 06	378	SHPADR	HEX 00010203040506
	379	*	
659B: 16	380	SHPLO	DFB SHAPES
659C: 2E	381		DFB SHAPES+\$18
659D: 46	382		DFB SHAPES+\$30
659E: 5E	383		DFB SHAPES+\$48
659F: 76	384		DFB SHAPES+\$60
65A0: 8E	385		DFB SHAPES+\$78
65A1: A6	386		DFB SHAPES+\$90
	387	*	
65A2: 3E	388	BSPLO	DFB BSHAPES
65A3: 4C	389		DFB BSHAPES+\$0E
65A4: 5A	390		DFB BSHAPES+\$1C
65A5: 68	391		DFB BSHAPES+\$2A
65A6: 76	392		DFB BSHAPES+\$38
65A7: 84	393		DFB BSHAPES+\$46
65A8: 92	394		DFB BSHAPES+\$54
65A9: A0	395		DFB BSHAPES+\$62
	396	DS	\$6C
	397	*SHAPE TABLE	GUN
6616: A0 81 00			
6619: A0 81 00			
661C: A0 81	398	SHAPES	HEX A08100A08100A081
661E: 00 A0 81			
6621: 00 A8 85			
6624: 00 A8	399		HEX 00A08100A88500A8

6626:	85 00 8A		
6629:	94 00 8A		
662C:	94 00	400	HEX 85008A94008A9400
		401	*2ND
662E:	00 85 00		
6631:	00 85 00		
6634:	00 85	402	HEX 0085000085000085
6636:	00 00 85		
6639:	00 A0 95		
663C:	00 A0	403	HEX 00008500A09500A0
663E:	95 00 A8		
6641:	D0 80 A8		
6644:	D0 80	404	HEX 9500A8D080A8D080
		405	*3RD
6646:	00 94 00		
6649:	00 94 00		
664C:	00 94	406	HEX 0094000094000094
664E:	00 00 94		
6651:	00 00 D5		
6654:	80 00	407	HEX 0000940000D58000
6656:	D5 80 A0		
6659:	C1 82 A0		
665C:	C1 82	408	HEX D580A0C182A0C182
		409	*4TH
665E:	00 D0 80		
6661:	00 D0 80		
6664:	00 D0	410	HEX 00D08000D08000D0
6666:	80 00 D0		
6669:	80 00 D4		
666C:	82 00	411	HEX 8000D08000D48200
666E:	D4 82 00		
6671:	85 8A 00		
6674:	85 8A	412	HEX D48200858A00858A
		413	*5TH
6676:	C0 82 00		
6679:	C0 82 00		
667C:	C0 82	414	HEX C08200C08200C082
667E:	00 C0 82		
6681:	00 D0 8A		
6684:	00 D0	415	HEX 00C08200D08A00D0
6686:	8A 00 94		
6689:	A8 00 94		
668C:	A8 00	416	HEX 8A0094A80094A800
		417	*6TH
668E:	00 8A 00		
6691:	00 8A 00		
6694:	00 8A	418	HEX 008A00008A00008A
6696:	00 00 8A		
6699:	00 C0 AA		
669C:	00 C0	419	HEX 00008A00C0AA00C0
669E:	AA 00 D0		
66A1:	A0 81 D0		
66A4:	A0 81	420	HEX AA00D0A081D0A081
		421	*7TH
66A6:	00 A8 00		
66A9:	00 A8 00		
66AC:	00 A8	422	HEX 00A80000A80000A8
66AE:	00 00 A8		
66B1:	00 00 AA		
66B4:	81 00	423	HEX 0000A80000AA8100

```

66B6: AA 81 CO
66B9: 82 85 CO
66BC: 82 85 424      HEX  AA81C08285C08285
        425      *
        426      DS   $80
        427 *BULLET SHAPE TABLE
673E: 40 01 40
6741: 01 40 01
6744: 40 428 BSHAPES HEX  40014001400140
6745: 01 40 01
6748: 40 01 40
674B: 01 429      HEX  01400140014001
        430 *2ND
674C: 00 06 00
674F: 06 00 06
6752: 00 431      HEX  00060006000600
6753: 06 00 06
6756: 00 06 00
6759: 06 432      HEX  06000600060006
        433 *3RD
675A: 00 18 00
675D: 18 00 18
6760: 00 434      HEX  00180018001800
6761: 18 00 18
6764: 00 18 00
6767: 18 435      HEX  18001800180018
        436 *4TH
6768: 00 60 00
676B: 60 00 60
676E: 00 437      HEX  00600060006000
676F: 60 00 60
6772: 00 60 00
6775: 60 438      HEX  60006000600060
        439 *5TH
6776: 00 03 00
6779: 03 00 03
677C: 00 440      HEX  00030003000300
677D: 03 00 03
6780: 00 03 00
6783: 03 441      HEX  03000300030003
        442 *6TH
6784: 00 0C 00
6787: 0C 00 0C
678A: 00 443      HEX  000C000C000C00
678B: 0C 00 0C
678E: 00 0C 00
6791: 0C 444      HEX  0C000C000C000C
        445 *7TH
6792: 00 30 00
6795: 30 00 30
6798: 00 446      HEX  00300030003000
6799: 30 00 30
679C: 00 30 00
679F: 30 447      HEX  30003000300030

```

--END ASSEMBLY--

ERRORS: 0

1952 BYTES

I'd like to emphasize that careful attention to detail is very important when programming. Machine language is very unforgiving. Failure to initialize a single variable could cause your graphics to go haywire. One of the most common mistakes is to clobber a register in your program or subroutine when calling another subroutine. Some programmers automatically save the Accumulator and X & Y registers by pushing them onto the stack before calling a subroutine, and restore them afterwards. It requires six instructions in each direction. Yet it makes more sense to have the called subroutine save the registers that it knows will be clobbered, and restore them before returning.

The setup routine for the drawing program is often a source for error. Although the setup is basically standard for a particular drawing subroutine, accidentally omitting one variable or failure to place a variable, in say, the Y register, can be disastrous. To give you an example of unexpected results, remove the STA TVERT in line 190 by NOPing the code in memory.

6169: EA EA EA

Run the program and watch the results. Imagine how long it might take to find this mistake. Debugging machine language graphics is difficult because events happen too quickly for the eye to detect. An Integer machine or an Integer ROM card with step and trace is almost a necessity. There have been times when I cleared the screen manually, set the graphics mode and put the machine in trace mode, so that I could watch the graphics being drawn in slow motion. Always remember to enter just after your CLRSCR or you will waste four or five minutes while the computer clears all 8K of Hi-Res memory. The commands for clearing screen #1 manually are as follows.

*2000: 00

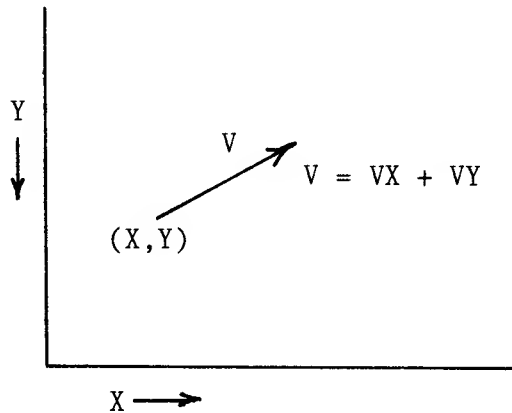
*2001<2000.3FFFM

Another debugging tool that is quite helpful is the single step debug module which is discussed on page xx. It allows you to step through each animation frame using the escape key. If your drawing routines are working as expected, single stepping will allow you to verify shape movement between successive frames.

STEERABLE SPACE SHIPS

The first game with a fully steerable space ship was developed at MIT. It was called Space War. While most of the newer computer owners won't recall this game, practically everyone is familiar with Asteroids. Most versions of this game have a steerable spaceship that can be thrust in the direction that it is headed. Although some versions invoke an automatic deceleration mode, some Asteroid games require the player to turn his ship around so that it thrusts in the opposite direction to slow down.

We previously demonstrated, with the topic of dropping bombs and shooting bullets, that objects move in the direction of their velocity vector.



An object's new position is its old position plus its change in position due to velocity, as shown:

$$\begin{aligned} X &= X + V_X \\ Y &= Y + V_Y. \end{aligned}$$

Using the Apple screen coordinate system for the example above, V_Y is negative and V_X is positive. Therefore,

$$\begin{aligned} X &= X + (V_X) \\ Y &= Y + (-V_Y) \end{aligned}$$

While the velocity vector may remain constant for many animation cycles, resulting in a ship moving in the same direction, sooner or later a new velocity vector will be inputted to change the object's course. This new velocity is the vector sum of the old velocity vector and the new velocity vector.

Those readers who have taken Physics will recall that a body's velocity changes due to external forces on it while it is in motion. In space ships, that

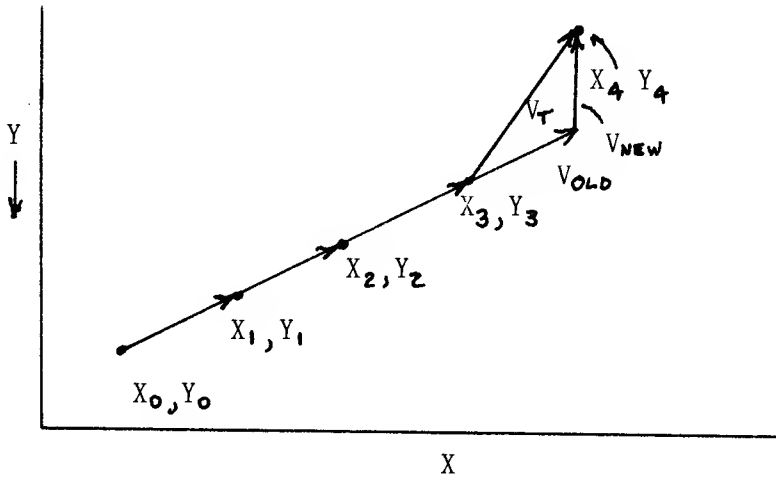
force is thrust. Thrust causes an acceleration of the object's mass as shown in the equation

$$F = m * a = m * \Delta V.$$

When thrust is applied to a space ship, it accelerates. If a ship is light and has a big engine with considerable thrust, it will accelerate quickly. But if it is heavy, it will accelerate much slower. This acceleration is essentially brought about by a change in the object's velocity if the object's mass is ignored.

Unless you are doing an actual simulation, in which values of thrust or force and an object's mass is important, only acceleration values need to be considered. Suitable values for arcade games are small and scaled, so that objects don't move too fast relative to their size, or fly off the screen in a blink of the eye.

If we consider a space ship that is in motion for two frames, then apply thrust during the third frame, it will change direction depending on the vector sum of its old and new velocity vectors. This is illustrated below. The applied thrust is straight upwards, so that $VX = 0$ and $VY = -2$. The ship's new velocity vector is calculated as follows:



$$VX = VX + VX = 2 + 0 = 2$$

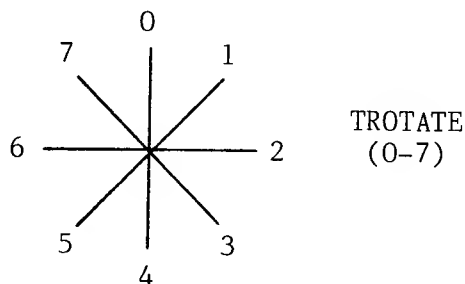
$$VY = VY + VY = -1 + (-2) = -3$$

The ship's new velocity vector causes it to move two units in the X direction and three in the negative Y direction during each frame until a new thrust vector is applied. The resultant position can be summarized in the table below.









FRAME	X	Y	VX	VY	
0	10	100	2	-1	$X = X + VX$
1	12	99	2	-1	$Y = Y + VY$
2	14	98	2	-1	
3	16	97	2	-3	Thrust applied here.
4	18	94	2	-3	
5	20	91			

A paddle will control the ship's direction in our simulation. The paddle's range (0-255) will be divided into eight directions (0-7). Dividing by 32 is simple in machine language. An arithmetic shift right (LSR, four times) will accomplish the task. After the division, paddle values 0-31 are equal to direction one, 32-63 to direction two, etc.

Now that we can control our ship in eight directions, we need shape tables for each of these directions. That means eight separate shapes. Rather than complicate matters unnecessarily, we will use a white ship and move it horizontally in one byte (7 pixel) increments, and vertically in eight line jumps. This way, we won't need extra sets of tables for the various offsets. Also, by conveniently keeping the shape within one of the 24 screen subsections, we can use an abbreviated set of YVERT tables.



PADDLE DIRECTION

			
0	1	2	3
			
4	5	6	7

The ship's thrust vector is completely dependent on the ship's paddle-controlled direction. If TROTATE, our paddle direction's value is four and the ship points down, it's thrust vector or velocity vector is $VX = 0$ and $VY = 1$. If TROTATE were seven, the ship points diagonally upward and to the left. The velocity vector is $VX = -1$ and $VY = -1$.

Note that many of our ship's directions produce negative velocity values, while others produce positive values. Separate routines are required for adding and subtracting in machine language. BASIC, however, just adds a negative number ($X = 5 + (-1)$). That's the clue. Adding a negative number is exactly the same as adding a positive number in machine language. Both use an ADC instruction. The difference is that negative numbers, like -1 , are represented by the two's complement which, for -1 , is $\$FF$. There is a limit for signed numbers of $+ \text{ or } -127$, because the BMI instruction tests the carry bit and considers the value negative if it is set.

If you add $\$FF$ to $\$03$, the result is $\$02$. Technically, the operation causes an overflow and the carry is set. But this doesn't concern us. With the simplification of our thrust vector addition problem, we can construct a table of velocity values for each TROTATE value.

THRUST VECTOR

	0	1	2	3	4	5	6	7
XT	00	01	01	01	00	FF	FF	FF
YT	FF	FF	00	01	01	01	00	FF

The thrust in this example is not cumulative. If the thrust button is on or pressed, the ship moves; if off, it stops. The ship drives like a car rather than floats, like it would in zero-gravity space. This is shown in the following:

$$XS = XS + XT \quad \text{and} \quad YS = YS + YT$$

where XS & YS is the ship's current position and XT & YT are the ship's velocity vector components.

With XT and YT both a function of TROTATE, the equations become:

$$XS = XS + XT(TROTATE) \text{ and } YS = YS + YT(TROTATE)$$

Thus, we can use table lookup to access the correct thrust for any ship direction.

```

LDX  TROTATE
CLC
LDA  XT,X      ;GET X THRUST VECTOR FOR TROTATE VALUE
ADC  XS        ;ADD TO X POSITION
STA  XS        ;STORE NEW VALUE

```

Now that the ship can be moved around the screen by both steering and thrusting, several tests must be implemented at the screen boundaries. Our Apple screen is 40 bytes wide by 24 subgroups deep. To index beyond the end of our tables would create unforeseen graphics, especially at the bottom of the screen.

XS can be tested for values greater than 39 and less than 0. In our case, with a ship moving only one position per frame, the test for less than 0 would be equal to the value FF or -1. If wrap-a-round is needed for an object leaving the right side of the screen, just set XS = 0 and it will reenter on the left. Likewise, setting XS = 39 works for objects leaving the left side of the screen. If the wrap-a-round effect is not desired, it requires setting XS = 39 for any attempt to leave the right side of the screen, and XS = 0 for any attempt to leave the left hand side of the screen. Essentially, the ship gets stuck at the edge. The boundary conditions at the top and bottom are similar.

Our drawing setup routine takes the paddle value into consideration to obtain the correctly rotated shape from the shape table for plotting. We can find the correct lo byte of the shape by the following formula:

$$\text{SHPL} = \text{SHPLO} (\text{TROTATE})$$

	SHPL	SHAPE TABLES		
0	\$74	\$6174	0th Shape	
1	\$7C	\$617C	1st Shape	
2	\$84	\$6184	2nd Shape	
3	\$8C	.	.	
	.	.	.	
	.	.	.	
7	\$AC	\$61AC	7th Shape	

YREG = TROTATE →

```

LDY  TROTATE    ;USE VALUE FOR DIRECTION OF ROTATED SHAPE
LDA  SHPLO,Y    ;AS INDEX TO PROPER LO BYTE OF SHAPE
STA  SHPL       ;STORE LO BYTE POINTER ON ZERO PAGE
LDA  #>SHAPES   ;GET HI BTE OF SHAPE TABLE
STA  SHPH       ;STORE IN ZERO PAGE

```

If the ship were turned so that it was pointing right, then TROTATE = 2 and SHPLO (2) = \$84. This lo byte of the shape table is stored as SHPL. The drawing routine will now plot the second shape from our shape table.

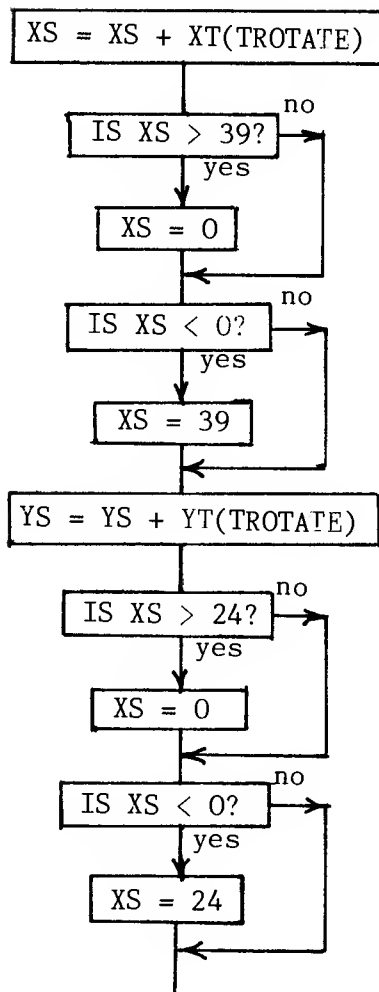
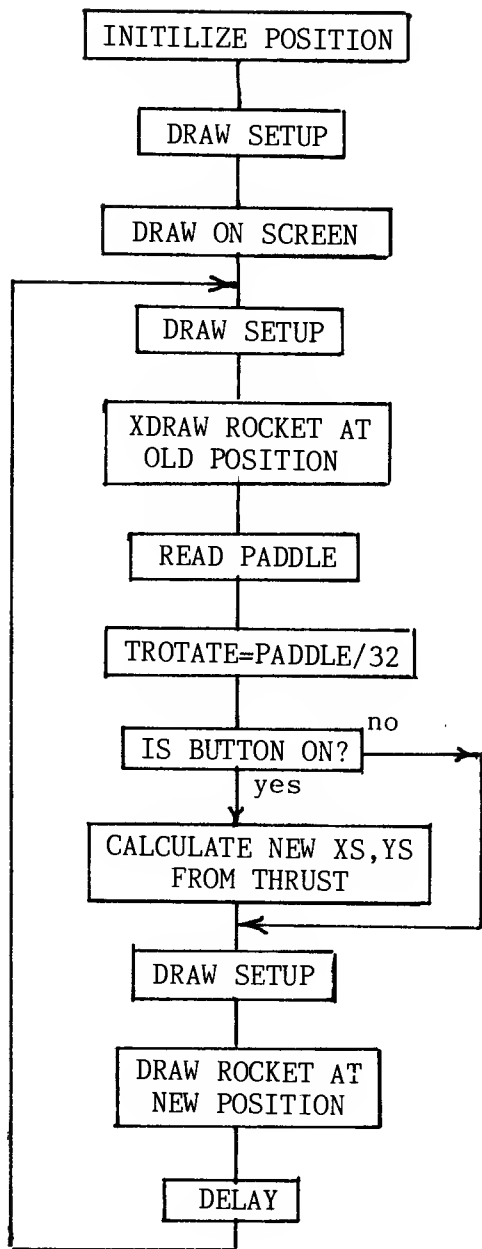
As we mentioned earlier, the ship is being moved eight lines at a time vertically to take advantage of plotting the ship within one of the 24 subsections on the Hi-Res screen. We can use the eight-line deep plotting routine, which was developed in the last chapter, if we don't cross any screen boundaries. This also simplifies and shortens our 192 element YVERT tables to two, 24 byte-long tables. Each table, one for the hi byte and one for the lo byte, stores the line address for the beginning of each of these blocks. The correct starting block for plotting our shape is a function of the ship's vertical position, YS (0-23). We index into the tables as before, using the Y register.

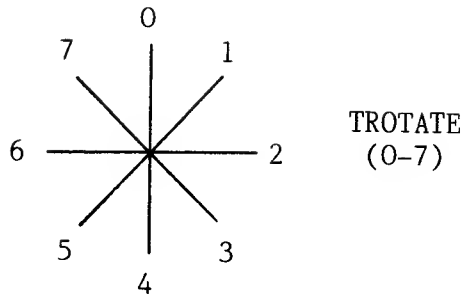
```

LDY  YS          ;SHIP'S VERTICAL POSITION (0-23)
LDA  YBLOCKL,Y   ;LOOK UP LO BYTE ADDRESS OF LINE
STA  HIRESL
LDA  YBLOCKH,Y   ;LOOK UP HI BYTE ADDRESS OF LINE
STA  HIRESH

```

Moving a space ship about the screen by paddle control is actually a simple case in the overall design of a game. One XDRAWS (erases) the ship at the old position, reads the paddle controller, calculates the ship's new position, and plots it at its new position. This is performed for each animation frame in an endless loop. Because the code is rather short, a considerable delay is needed to slow down the animation frame rate. With very short delays in the monitor delay subroutine, the frame rate exceeds the 30 frame-per-second scan rate of the television. The ship appears to blink at random during its movement. The television hasn't finished drawing the first animation cycle while you moved your ship two or three times in between. A longer delay, wherein the WAIT subroutine has a value of \$C0 to \$FF in the Accumulator, works fine. The flow chart of this steerable rocket program is shown below.



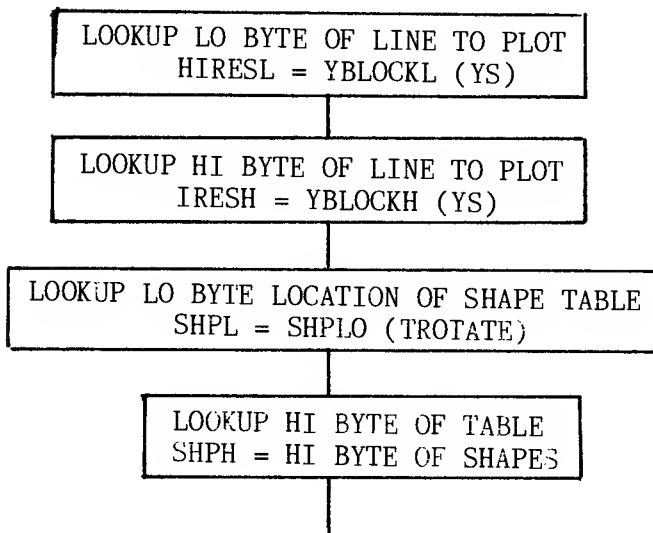


PADDLE DIRECTION

THRUST VECTOR

	0	1	2	3	4	5	6	7
XT	00	01	01	01	00	FF	FF	FF
YT	FF	FF	00	01	01	01	00	FF

DRAWING SETUP



```

1      *ROCKET (DRIVES LIKE CAR)
2      ORG    $6000
6000: 4C 09 60 3      JMP    PROG
4      XS      DS    1
5      YS      DS    1
6      PDL     DS    1
7      LNGH    DS    1
8      ROTATE  DS    1
9      TROTATE DS    1
10     HIRESL   EQU   $FB
11     HIRESH   EQU   HIRESL+$1
12     SHPL     EQU   $FD
13     SHPH     EQU   SHPL+$1
14     PREAD    EQU   $FB1E
15     *ENTER HERE FIRST TIME ACCESS
6009: AD 50 C0 16     PROG    LDA    $C050
600C: AD 52 C0 17         LDA    $C052
600F: AD 57 C0 18         LDA    $C057
6012: 20 13 61 19         JSR    CLRSCR
20     *INITILIZE ROCKET'S STARTING POSITION
6015: A9 14 21         LDA    #$14
6017: 8D 03 60 22         STA    XS
601A: A9 0A 23         LDA    #$0A
601C: 8D 04 60 24         STA    YS
601F: A9 00 25         LDA    #$00
6021: 8D 07 60 26         STA    ROTATE
6024: 20 F6 60 27         JSR    DSETUP
6027: 20 CF 60 28         JSR    DRAW          ;DRAW INITIAL POSITION ROCKET
29     * PADDLE READ
602A: 20 F6 60 30     START   JSR    DSETUP
602D: 20 CF 60 31         JSR    DRAW          ;ERASE ROCKET
6030: A2 01 32         LDX    #$01
6032: 20 1E FB 33         JSR    PREAD
6035: C0 F9 34         CPY    #$F9          ;CLIP VALUE (0-250)
6037: 90 02 35         BLT    SKIPP
6039: A0 F8 36         LDY    #$F8
603B: 8C 05 60 37     SKIPP   STY    PDL
603E: 98 38         TYA
603F: CD 07 60 39         CMP    ROTATE      ;PADDLE<ROTATE POS THEN SUBTRACT 5
6042: B0 1B 40         BGE    PADDLE3
6044: AD 07 60 41         LDA    ROTATE
6047: 38 42         SEC
6048: E9 05 43         SBC    #$05
604A: B0 05 44         BGE    PADDLE1      ;MAKE SURE =>0
604C: A9 00 45         LDA    #$00
604E: 8D 07 60 46         STA    ROTATE
6051: CD 05 60 47     PADDLE1  CMP    PDL          ;DON'T WANT TO GO PAST PADDLE POS
6054: B0 03 48         BGE    PADDLE2
6056: AD 05 60 49         LDA    PDL
6059: 8D 07 60 50     PADDLE2  STA    ROTATE
605C: 4C 72 60 51         JMP    PADDLE5
605F: CD 07 60 52     PADDLE3  CMP    ROTATE      ;PADDLE>ROTATE POS THEN ADD 5
6062: F0 0B 53         BEQ    PADDLE4
6064: AD 07 60 54         LDA    ROTATE
6067: 18 55         CLC
6068: 69 05 56         ADC    #$05
606A: CD 05 60 57         CMP    PDL          ;DON'T WANT TO GO PAST PADDLE POS
606D: 90 03 58         BLT    PADDLE5
606F: AD 05 60 59     PADDLE4  LDA    PDL
6072: 8D 07 60 60     PADDLE5  STA    ROTATE

```


6075:	4A	61	LSR		
6076:	4A	62	LSR		
6077:	4A	63	LSR		
6078:	4A	64	LSR		
6079:	4A	65	LSR		
607A:	8D 08 60	66	STA	TROTATE	
		67	*		
607D:	AD 62 C0	68	LDA	\$C062	;NEG IF BUTTON PRESSED
6080:	30 03	69	BMI	THRUST	
6082:	4C C0 60	70	JMP	NOTHRUST	
6085:	AE 08 60	71	LDX	TROTATE	
6088:	18	72	CLC		
6089:	BD 5D 61	73	LDA	XT,X	;GET X THRUST VECTOR
608C:	6D 03 60	74	ADC	XS	;ADD TO X POSITION
608F:	C9 28	75	CMP	#\$28	;CHECK IF OFF SCREEN RT
6091:	D0 08	76	BNE	NWRAP1	;O.K.
6093:	A9 00	77	LDA	#\$00	;NO! THEN WRAP-A-ROUND
6095:	8D 03 60	78	STA	XS	
6098:	4C A4 60	79	JMP	NOWY	
609B:	C9 FF	80	NWRAP1	CMP	#\$FF ;LESS THAN 0? (-1)
609D:	D0 02	81	BNE	NWRAP2	;O.K.
609F:	A9 27	82	LDA	#\$27	;NO! THEN WRAP-A-ROUND
60A1:	8D 03 60	83	NWRAP2	STA	XS
60A4:	18	84	NOWY	CLC	
60A5:	BD 65 61	85	LDA	YT,X	;GET Y THRUST VECTOR
60A8:	6D 04 60	86	ADC	YS	;ADD TO Y POSITION
60AB:	C9 18	87	CMP	#\$18	;CHECK IF OFF SCREEN BOTTOM
60AD:	D0 08	88	BNE	NWRAP3	;O.K.
60AF:	A9 00	89	LDA	#\$00	;NO! THEN WRAP-A-ROUND
60B1:	8D 04 60	90	STA	YS	
60B4:	4C C0 60	91	JMP	NOTHRUST	
60B7:	C9 FF	92	NWRAP3	CMP	#\$FF ;LESS THAN 0? (-1)
60B9:	D0 02	93	BNE	NWRAP4	;O.K.
60BB:	A9 17	94	LDA	#\$17	;NO! THEN WRAP-A-ROUND
60BD:	8D 04 60	95	NWRAP4	STA	YS
60C0:	EA	96	NOTHRUST	NOP	
		97	*		
60C1:	20 F6 60	98	JSR	DSETUP	
60C4:	20 CF 60	99	JSR	DRAW	;DRAW ROCKET
60C7:	A9 70	100	LDA	#\$70	
60C9:	20 A8 FC	101	JSR	\$FCA8	; SHORT DELAY
60CC:	4C 2A 60	102	JMP	START	
		103	*SUBROUTINE TO DRAW ROCKET 1 BYTE BY 8 ROWS		
60CF:	A2 00	104	DRAW	LDX	#\$00
60D1:	A9 01	105	LDA	#\$01	
60D3:	8D 06 60	106	STA	LNGH	
60D6:	A1 FD	107	DRAW2	LDA	(SHPL,X) ;GET BYTE FROM SHAPE TABLE
60D8:	51 FB	108	EOR	(HIRESL),Y	
60DA:	91 FB	109	STA	(HIRESL),Y	;PUT ON HIRES SCREEN
60DC:	A5 FC	110	LDA	HIRESH	
60DE:	18	111	CLC		
60DF:	69 04	112	ADC	#\$04	;THIS GETS TO NEXT ROW IN BLOCK
60E1:	85 FC	113	STA	HIRESH	
60E3:	E6 FD	114	INC	SHPL	;NEXT BYTE OF SHAPE TABLE
60E5:	C9 40	115	CMP	#\$40	;ARE WE FINISHED WITH 8 ROWS
60E7:	90 ED	116	BCC	DRAW2	;NO DO NEXT BYTE
60E9:	E9 20	117	SBC	#\$20	;RETURN TO TOP ROW
60EB:	85 FC	118	STA	HIRESH	
60ED:	CE 06 60	119	DEC	LNGH	
60F0:	F0 03	120	BEQ	DRAW3	;FINISHED?

```

60F2: C8      121      INY                      ;NEXT COLUMN OF 8 ROWS
60F3: DO E1   122      BNE  DRAW2
60F5: 60      123      DRAW3      RTS
                        124      *DRAWING SETUP SUBROUTINE
60F6: AC 04 60 125      DSETUP    LDY  YS                      ;SHIP'S VERTICAL POS (0-23)
60F9: B9 45 61 126      LDA  YBLOCKL,Y  ;LOOK UP LO BYTE OF LINE
60FC: 85 FB   127      STA  HIRESL
60FE: B9 2D 61 128      LDA  YBLOCKH,Y  ;LOOK UP HI BYTE OF LINE
6101: 85 FC   129      STA  HIRESH
6103: AC 08 60 130      LDY  TROTATE
6106: B9 6D 61 131      LDA  SHPLO,Y
6109: 85 FD   132      STA  SHPL
610B: A9 61   133      LDA  #>SHAPES
610D: 85 FE   134      STA  SHPH
610F: AC 03 60 135      LDY  XS                      ;DISPLACEMENT INTO LINE
6112: 60      136      RTS
                        137      *CLEAR SCREEN SUBROUTINE
6113: A9 00   138      CLRSCR    LDA  #$00
6115: 85 FB   139      STA  HIRESL
6117: A9 20   140      LDA  #$20
6119: 85 FC   141      STA  HIRESH
611B: A0 00   142      CLR1      LDY  #$00
611D: A9 00   143      LDA  #$00
611F: 91 FB   144      CLR2      STA  (HIRESL),Y
6121: C8      145      INY
6122: DO FB   146      BNE  CLR2
6124: E6 FC   147      INC  HIRESH
6126: A5 FC   148      LDA  HIRESH
6128: C9 40   149      CMP  #$40
612A: 90 EF   150      BCC  CLR1
612C: 60      151      RTS
                        152      *TABLES OF STARTING VALUE OF EACH OF 24 BLOCKS
612D: 20 20 21
6130: 21 22 22
6133: 23 23 20
6136: 20      153      YBLOCKH  HEX  20202121222223232020
6137: 21 21 22
613A: 22 23 23
613D: 20 20 21
6140: 21      154      HEX  21212222232320202121
6141: 22 22 23
6144: 23      155      HEX  22222323
6145: 00 80 00
6148: 80 00 80
614B: 00 80 28
614E: A8      156      YBLOCKL  HEX  008000800080008028A8
614F: 28 A8 28
6152: A8 28 A8
6155: 50 DO 50
6158: DO      157      HEX  28A828A828A850DO50DO
6159: 50 DO 50
615C: DO      158      HEX  50DO50DO
                        159      *TABLES OF DIRECTION VECTORS FOR 8 ROTATION VALUES
615D: 00 01 01
6160: 01 00 FF
6163: FF FF   160      XT      HEX  0001010100FFFFFF
6165: FF FF 00
6168: 01 01 01
616B: 00 FF   161      YT      HEX  FFFF0001010100FF

```

```

162 *GENERATE SHPLO TABLE
163 *( INDEX TO LO BYTE OF EACH ROCKET SHAPE)
616D: 75      164 SHPLO      DFB  SHAPES
616E: 7D      165          DFB  SHAPES+$08
616F: 85      166          DFB  SHAPES+$10
6170: 8D      167          DFB  SHAPES+$18
6171: 95      168          DB   SHAPES+$20
6172: 9D      169          DFB  SHAPES+$28
6173: A5      170          DFB  SHAPES+$30
6174: AD      171          DFB  SHAPES+$38
172 *
173 *ROCKET SHAPES
6175: 00 08 08
6178: 08 1C 1C
617B: 36 00      174 SHAPES      HEX  000808081C1C3600
175 *2ND
617D: 00 00 20
6180: 14 0F 1C
6183: 08 08      176          HEX  000020140F1C0808
177 *3RD
6185: 00 00 02
6188: 0E 7C 0E
618B: 02 00      178          HEX  0000020E7C0E0200
179 *4TH
618D: 00 08 08
6190: 1C 0F 14
6193: 20 00      180          HEX  0008081C0F142000
181 *5TH
6195: 00 00 36
6198: 1C 1C 08
619B: 08 08      182          HEX  0000361C1C080808
183 *6TH
619D: 00 08 08
61A0: 1C 78 14
61A3: 02 00      184          HEX  0008081C78140200
185 *7TH
61A5: 00 00 20
61A8: 38 1F 38
61AB: 20 00      186          HEX  000020381F382000
187 *8TH
61AD: 00 00 02
61B0: 14 78 1C
61B3: 08 08      188          HEX  00000214781C0808

```

--END ASSEMBLY-- 437 BYTES

STEERABLE & FREE FLOATING

Objects in the real world, once started in motion, tend to remain in motion. Isaac Newton stated it more formally in his first law of motion. Objects remain at rest or in motion along a straight line unless a force is applied on them to change that motion. The force in most games is thrust.

In the last section, we dealt with a spaceship that had a velocity only when thrust was applied to it. We avoided any sustained velocity by zeroing our velocity vector when there was no thrust. Normally, the equations for determining the velocity and position of an object in motion are as follows (They were discussed briefly under the section on bullets and bomb drops.):

$$\begin{array}{llllll}
 V_{NEW} & = & V_{OLD} & + & \Delta V & \Delta V & = & \text{CHANGE IN VELOCITY} \\
 D_{NEW} & = & D_{OLD} & + & \Delta D & \Delta D & = & \text{CHANGE IN POSITION} \\
 & & & & & & & \text{OVER AN ANIMATION} \\
 & & & & & & & \text{FRAME DUE} \\
 & & & & & & & \text{TO VELOCITY} \\
 \text{OR} & & & & & & & \\
 D_{NEW} & = & D_{OLD} & + & V_{NEW} & & &
 \end{array}$$

This breaks down into components in the X and Y directions.

$$VX_{NEW} = VX_{OLD} + \Delta VX$$

$$VY_{NEW} = VY_{OLD} + \Delta VY$$

$$X_{NEW} = X_{OLD} + VX$$

$$Y_{NEW} = Y_{OLD} + VY$$

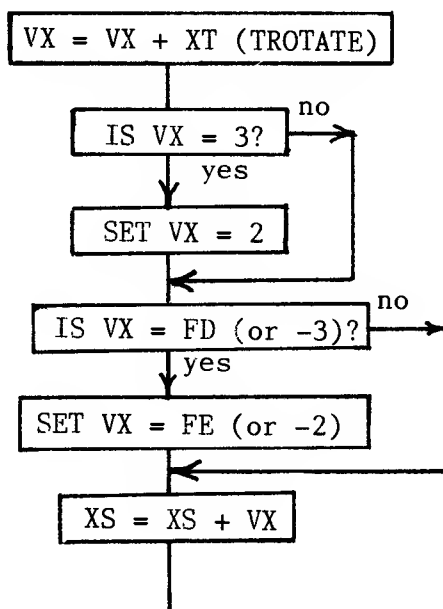
Now, when an object is thrust in any direction, the increase in velocity is cumulative. For example, if thrust were applied in the positive X direction with a force of 1 unit/ frame, the new VX would increase from zero by units of one for each animation frame.

	CYCLE	VX	X		CYCLE	VY	Y
VX = 1	0	0	0	similarly VY = 2	0	0	0
	1	1	1		1	2	2
	2	2	3		2	4	6
	3	3	6		3	6	12
	4	4	10		4	8	20

It becomes clear from our example that if you accelerate for too many animation frames, the space ship will be moving fairly fast. While the amount of relative movement depends on your choice of scale, the ship moves to the left or right seven pixels for every unit change instead of by individual pixels. If, by

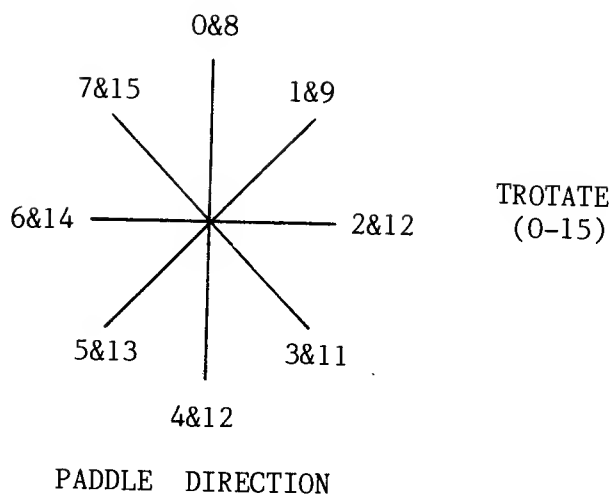
the fourth frame, our velocity were 4 units/frame, we would actually be moving 28 pixels horizontally per frame. With a slow program, framing at 10 frames/second, the ship would move entirely across the screen in 1 second. More likely, with faster animation, it would take less than half a second. This may be too fast.

A speed brake can be incorporated into the algorithm to prevent the velocity from exceeding a preset value. This would be analogous to wind resistance on a fast moving automobile. It prevents a vehicle from reaching ever-increasing speeds. I chose a maximum velocity of 2 units/ frame. It was an arbitrary choice based on keeping the animation smooth. Discontinuous jumps at higher velocities produced degraded animation. The brake is placed just after the velocity equations. If the value of VX or VY exceeds 2 units/frame, it is trimmed back to 2 units/frame.



The flow chart, as shown for the X direction (horizontal), is relatively straight-forward. Again, the velocity vector is a function of the ship's paddle-controlled direction.

The paddle control in the non-free-floating ship was restrictive. It prevented you from directly reaching the straight-up position (0) from a position pointing upwards and to the left (7). When the paddle's value was divided by 32, giving TROTATE values 0-7, it lacked wrap-a-round capability. It would be better to be able to turn the ship nearly twice around with one twist of the paddle. This is accomplished by dividing the paddle reading by 16. This gives TROTATE values 0-15.

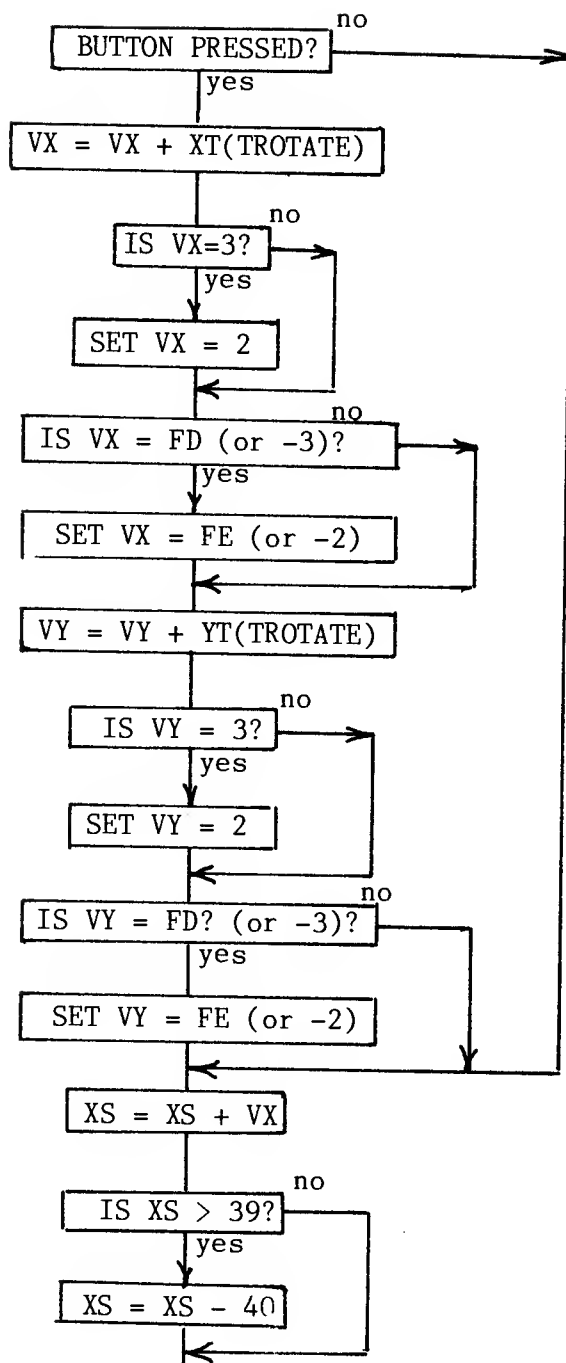


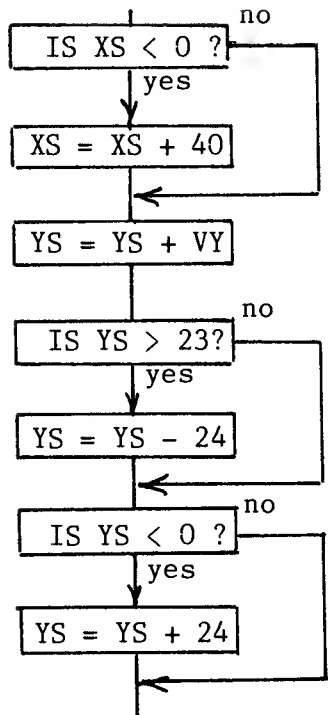
THRUST VECTOR

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
XT	01	01	01	01	00	FF	FF	FF	00	01	01	01	00	FF	FF	FF
YT	FF	FF	00	01	01	01	00	FF	FF	FF	00	01	01	01	00	FF

Since the proper shape is drawn from the correct section of the shape table by setting the appropriate lo and hi byte pointers for that shape, the index to these pointers must be corrected for the extra number of rotation angles. With TROTATE doubled to 16 values, the SHPLO table, which contains the 16 pointers to each shape, must also contain 16 values. Since TROTATE values are duplicated after 8 values, the SHPLO table, as well as the XT and YT tables, are duplicated after eight values.

Except for the changes discussed above, the steerable and free-floating ship routine is much like the former routine, in which the ship drives around like a car. The flow chart and code are shown below. It might be instructive to change the delay in line #129 to a small value like \$05 to see what happens when the animation frame rate exceeds the television's scan rate.





```

1      *ROCKET (FREE FLOATING)
2      ORG $6000
6000: 4C 0B 60 3      JMP PROG
4      XS          DS 1
5      YS          DS 1
6      VX          DS 1
7      VY          DS 1
8      PDL         DS 1
9      LNGH        DS 1
10     ROTATE      DS 1
11     TROTATE     DS 1
12     HIRESL      EQU $FB
13     HIRESH      EQU HIRESL+$1
14     SHPL        EQU $FD
15     SHPH        EQU SHPL+$1
16     PREAD       EQU $FB1E
17     *ENTER HERE FIRST TIME ACCESS
600B: AD 50 C0 18     PROG LDA $C050
600E: AD 52 C0 19     LDA $C052
6011: AD 57 C0 20     LDA $C057
6014: 20 49 61 21     JSR CLRSCR
22     *INITILIZE ROCKET'S STARTING POSITION
6017: A9 14 23       LDA #$14
6019: 8D 03 60 24     STA XS
601C: A9 0A 25       LDA #$0A

```


601E:	8D	04	60	26		STA	YS	
6021:	A9	00		27		LDA	#\$00	
6023:	8D	05	60	28		STA	VX	
6026:	8D	06	60	29		STA	VY	
6029:	8D	09	60	30		STA	ROTATE	
602C:	20	2C	61	31		JSR	DSETUP	
602F:	20	05	61	32		JSR	DRAW	
				33	* PADDLE READ			
6032:	20	2C	61	34	START	JSR	DSETUP	
6035:	20	05	61	35		JSR	DRAW	
6038:	A2	01		36		LDX	#\$01	
603A:	20	1E	FB	37		JSR	PREAD	
603D:	C0	F9		38		CPY	#\$F9	;CLIP VALUE (0-250)
603F:	90	02		39		BLT	SKIPP	
6041:	A0	F8		40		LDY	#\$F8	
6043:	8C	07	60	41	SKIPP	STY	PDL	
6046:	98			42		TYA		
6047:	CD	09	60	43		CMP	ROTATE	;PADDLE<ROTATE POS THEN SUBTRACT 5
604A:	B0	1B		44		BGE	PADDLE3	
604C:	AD	09	60	45		LDA	ROTATE	
604F:	38			46		SEC		
6050:	E9	05		47		SBC	#\$05	
6052:	B0	05		48		BGE	PADDLE1	;MAKE SURE =>0
6054:	A9	00		49		LDA	#\$00	
6056:	8D	09	60	50		STA	ROTATE	
6059:	CD	07	60	51	PADDLE1	CMP	PDL	;DON'T WANT TO GO PAST PADDLE POS
605C:	B0	03		52		BGE	PADDLE2	
605E:	AD	07	60	53		LDA	PDL	
6061:	8D	09	60	54	PADDLE2	STA	ROTATE	
6064:	4C	7A	60	55		JMP	PADDLE5	
6067:	CD	09	60	56	PADDLE3	CMP	ROTATE	;PADDLE>ROTATE POS THEN ADD 5
606A:	F0	0B		57		BEQ	PADDLE4	
606C:	AD	09	60	58		LDA	ROTATE	
606F:	18			59		CLC		
6070:	69	05		60		ADC	#\$05	
6072:	CD	07	60	61		CMP	PDL	;DON'T WANT TO GO PAST PADDLE POS
6075:	90	03		62		BLT	PADDLE5	
6077:	AD	07	60	63	PADDLE4	LDA	PDL	
607A:	8D	09	60	64	PADDLE5	STA	ROTATE	
607D:	4A			65		LSR		;DIVIDE BY 16 TO GET ROTATION(0-15)
607E:	4A			66		LSR		;-(OR TWO ROATIONS AROUND)
607F:	4A			67		LSR		
6080:	4A			68		LSR		
6081:	8D	0A	60	69		STA	TROTATE	
				70	*			
6084:	AD	62	C0	71		LDA	\$C062	;NEG IF BUTTON PRESSED
6087:	30	03		72		BMI	THRUST	
6089:	4C	C1	60	73		JMP	NOTHRUST	
608C:	AE	0A	60	74	THRUST	LDX	TROTATE	
				75	*UPDATE VELOCITY VX AND VY			
608F:	18			76		CLC		
6090:	BD	93	61	77		LDA	XT,X	;GET X THRUST VECTOR
6093:	6D	05	60	78		ADC	VX	
6096:	C9	FD		79		CMP	#\$FD	
6098:	D0	05		80		BNE	NOCLIP	
609A:	A9	FE		81		LDA	#\$FE	
609C:	4C	A5	60	82		JMP	NOCLIP1	
609F:	C9	03		83	NOCLIP	CMP	#\$03	;CLIP MAX VELOCITY AT 2
60A1:	D0	02		84		BNE	NOCLIP1	
60A3:	A9	02		85		LDA	#\$02	

```

60A5: 8D 05 60 86 NOCLIP1 STA VX ;STORE X VELOCITY
60A8: 18 87 CLC
60A9: BD A3 61 88 LDA YT,X
60AC: 6D 06 60 89 ADC VY
60AF: C9 FD 90 CMP #$FD
60B1: D0 05 91 BNE NOCLIP2
60B3: A9 FE 92 LDA #$FE
60B5: 4C BE 60 93 JMP NOCLIP3
60B8: C9 03 94 NOCLIP2 CMP #$03 ;CLIP MAX VELOCITY AT 2
60BA: D0 02 95 BNE NOCLIP3
60BC: A9 02 96 LDA #$02
60BE: 8D 06 60 97 NOCLIP3 STA VY ;STORE Y VELOCITY
98 *UPDATE SHIP'S X POSITION XS
60C1: 18 99 NOTHRUST CLC
60C2: AD 05 60 100 LDA VX
60C5: 6D 03 60 101 ADC XS
60C8: C9 E0 102 CMP #$E0 ;CHECK FOR WRAPAROUND LEFT
60CA: 90 06 103 BLT NWRAP1
60CC: 18 104 CLC
60CD: 69 28 105 ADC #$28 ;FIX BY ADDING 40
60CF: 4C D9 60 106 JMP NWRAP2
60D2: C9 28 107 NWRAP1 CMP #$28 ;CHECK FOR WRAPAROUND RIGHT
60D4: 90 03 108 BLT NWRAP2
60D6: 38 109 SEC
60D7: E9 28 110 SBC #$28 ;FIX BY SUBTRACTING 40
60D9: 8D 03 60 111 NWRAP2 STA XS ;STORE SHIP'S NEW X POS
112 *UPDATE SHIP'S Y POSITION YS
60DC: 18 113 CLC
60DD: AD 06 60 114 LDA VY
60E0: 6D 04 60 115 ADC YS
60E3: C9 E0 116 CMP #$E0 ;CHECK FOR WRAPAROUND TOP
60E5: 90 06 117 BLT NWRAP3
60E7: 18 118 CLC
60E8: 69 18 119 ADC #$18 ;FIX BY ADDING 24
60EA: 4C F4 60 120 JMP NWRAP4
60ED: C9 18 121 NWRAP3 CMP #$18 CHECK FOR WRAPAROUND BOTTOM
60EF: 90 03 122 BLT NWRAP4
60F1: 38 123 SEC
60F2: E9 18 124 SBC #$18 ;FIX BY SUBTRACTING 24
60F4: 8D 04 60 125 NWRAP4 STA YS ;STORE NEW Y POSITION
126 *
60F7: 20 2C 61 127 JSR DSETUP
60FA: 20 05 61 128 JSR DRAW
60FD: A9 C0 129 LDA #$C0
60FF: 20 A8 FC 130 JSR $FCA8 ;SHORT DELAY
6102: 4C 32 60 131 JMP START
132 *SUBROUTINE TO DRAW ROCKET 1 BYTEBY 8 ROWS
6105: A2 00 133 DRAW LDX #$00
6107: A9 01 134 LDA #$01
6109: 8D 08 60 135 STA LNGL
610C: A1 FD 136 DRAW2 LDA (SHPL,X) ;GET BYTE FROM SHAPE TABLE
610E: 51 FB 137 EOR (HIRESL),Y
6110: 91 FB 138 STA (HIRESL),Y ;PUT ON HIRES SCREEN
6112: A5 FC 139 LDA HIRESH
6114: 18 140 CLC
6115: 69 04 141 ADC #$04 ;THIS GETS TO NEXT ROW IN BLOCK
6117: 85 FC 142 STA HIRESH
6119: E6 FD 143 INC SHPL ;NEXT BYTE OF SHAPE TABLE
611B: C9 40 144 CMP #$40 ;ARE WE FINISHED WITH 8 ROWS
611D: 90 ED 145 BCC DRAW2 ;NO DO NEXT BYTE

```

611F:	E9 20	146	SBC	#\$20	;RETURN TO TOP ROW
6121:	85 FC	147	STA	HIRESH	
6123:	CE 08 60	148	DEC	LNGH	
6126:	F0 03	149	BEQ	DRAW3	;FINISHED?
6128:	C8	150	INY		;NEXT COLUMN OF 8 ROWS
6129:	D0 E1	151	BNE	DRAW2	
612B:	60	152	DRAW3	RTS	
		153	*DRAWING	SETUP SUBROUTINE	
612C:	AC 04 60	154	DSETUP	LDY	YS
612F:	B9 7B 61	155	LDA	YBLOCKL,Y	;LOOK UP LO BYTE OF LINE
6132:	85 FB	156	STA	HIRESL	
6134:	B9 63 61	157	LDA	YBLOCKH,Y	
6137:	85 FC	158	STA	HIRESH	
6139:	AC 0A 60	159	LDY	TROTATE	
613C:	B9 B3 61	160	LDA	SHPLO,Y	
613F:	85 FD	161	STA	SHPL	
6141:	A9 62	162	LDA	#>SHAPES	
6143:	85 FE	163	STA	SHPH	
6145:	AC 03 60	164	LDY	XS	;DISPLACEMENT INTO LINE
6148:	60	165	RTS		
		166	*CLEAR SCREEN	SUBROUTINE	
6149:	A9 00	167	CLRSCR	LDA	#\$00
614B:	85 FB	168	STA	HIRESL	
614D:	A9 20	169	LDA	#\$20	
614F:	85 FC	170	STA	HIRESH	
6151:	A0 00	171	CLR1	LDY	#\$00
6153:	A9 00	172	LDA	#\$00	
6155:	91 FB	173	CLR2	STA	(HIRESL),Y
6157:	C8	174	INY		
6158:	D0 FB	175	BNE	CLR2	
615A:	E6 FC	176	INC	HIRESH	
615C:	A5 FC	177	LDA	HIRESH	
615E:	C9 40	178	CMP	#\$40	
6160:	90 EF	179	BCC	CLR1	
6162:	60	180	RTS		
		181	*TABLES OF STARTING VALUE OF EACH OF 20 BLOCKS		
6163:	20 20 21				
6166:	21 22 22				
6169:	23 23 20				
616C:	20	182	YBLOCKH	HEX	20202121222223232020
616D:	21 21 22				
6170:	22 23 23				
6173:	20 20 21				
6176:	21	183		HEX	21212222232320202121
6177:	22 22 23				
617A:	23	184		HEX	22222323
617B:	00 80 00				
617E:	80 00 80				
6181:	00 80 28				
6184:	A8	185	YBLOCKL	HEX	008000800080008028A8
6185:	28 A8 28				
6188:	A8 28 A8				
618B:	50 D0 50				
618E:	D0	186		HEX	28A828A828A850D050D0
618F:	50 D0 50				
6192:	D0	187		HEX	50D050D0
		188	*		
6193:	00 01 01				
6196:	01 00 FF				
6199:	FF FF	189	XT	HEX	0001010100FFFFF

619B:	00 01 01			
619E:	01 00 FF			
61A1:	FF FF	190	HEX	0001010100FFFFFF
61A3:	FF FF 00			
61A6:	01 01 01			
61A9:	00 FF	191	YT	HEX FFFF0001010100FF
61AB:	FF FF 00			
61AE:	01 01 01			
61B1:	00 FF	192	HEX	FFFF0001010100FF
		193	*	
61B3:	13	194	SHPLO	DFB SHAPES
61B4:	1B	195		DFB SHAPES+\$08
61B5:	23	196		DFB SHAPES+\$10
61B6:	2B	197		DFB SHAPES+\$18
61B7:	33	198		DFB SHAPES+\$20
61B8:	3B	199		DFB SHAPES+\$28
61B9:	43	200		DFB SHAPES+\$30
61BA:	4B	201		DFB SHAPES+\$38
		202	*NEXT GROUP BECAUSE PADDLE (0-15) INDEXES	
		203	*INTO SHAPE TABLE TWICE	
61BB:	13	204		DFB SHAPES
61BC:	1B	205		DFB SHAPES+\$08
61BD:	23	206		DFB SHAPES+\$10
61BE:	2B	207		DFB SHAPES+\$18
61BF:	33	208		DFB SHAPES+\$20
61C0:	3B	209		DFB SHAPES+\$28
61C1:	43	210		DFB SHAPES+\$30
61C2:	4B	211		DFB SHAPES+\$38
		212	*	
		213	SPACE	DS 80
		214	*ROCKET SHAPES	
6213:	00 08 08			
6216:	08 1C 1C			
6219:	36 00	215	SHAPES	HEX 000808081C1C3600
		216	*2ND	
621B:	00 00 20			
621E:	14 0F 1C			
6221:	08 08	217		HEX 000020140F1C0808
		218	*3RD	
6223:	00 00 02			
6226:	0E 7C 0E			
6229:	02 00	219		HEX 0000020E7C0E0200
		220	*4TH	
622B:	00 08 08			
622E:	1C 0F 14			
6231:	20 00	221		HEX 0008081C0F142000
		222	*5TH	
6233:	00 00 36			
6236:	1C 1C 08			
6239:	08 08	223		HEX 0000361C1C080808
		224	*6TH	
623B:	00 08 08			
623E:	1C 78 14			
6241:	02 00	225		HEX 0008081C78140200
		226	*7TH	
6243:	00 00 20			
6246:	38 1F 38			
6249:	20 00	227		HEX 000020381F382000
		228	*8TH	

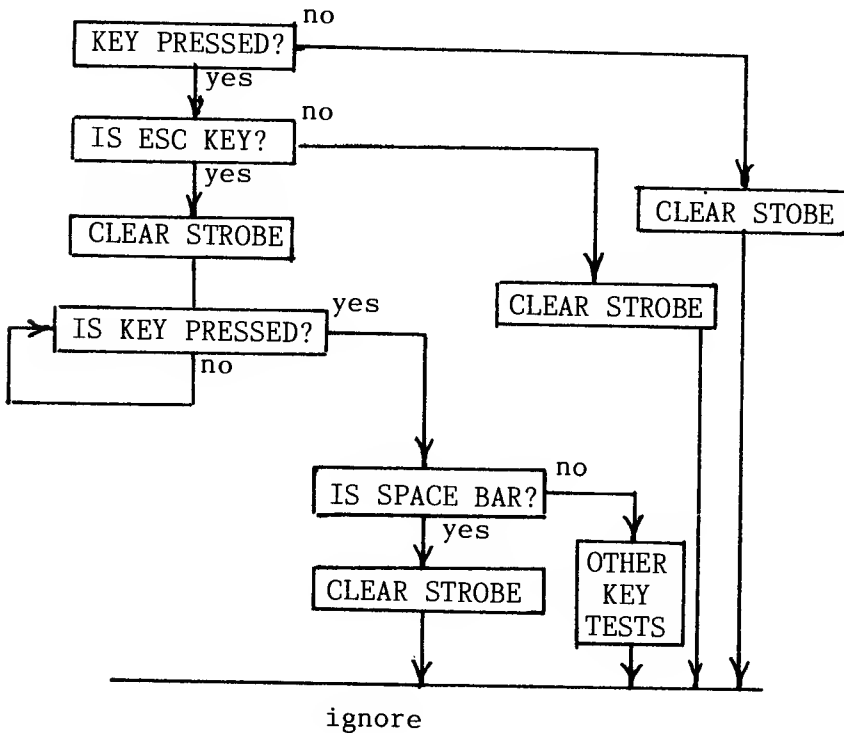
624B: 00 00 02
624E: 14 78 1C
6251: 08 08 229

HEX 00000214781C0808

--END ASSEMBLY-- 595 BYTES

DEBUG PACKAGE

The debug package that was mentioned earlier is a very useful tool for programmers. It allows you to single step animation by stopping the animation with the ESC key. Once the ESC key is pressed, the program goes into a tight loop while waiting for another key press. Any key except the ESC key will release it. But since every key, with the exception of the space bar, fails to clear the keyboard strobe, the computer thinks a key has been pressed when it encounters the debug subroutine during the next animation frame. Of course, if the key last pressed was the ESC, it will be caught in that small loop once again, and stop or single step. Yet if it is another key, it won't stop the animation, but would proceed to other tests in the package. The space bar would release it totally from the subroutine by clearing the keyboard strobe.



The debug package is designed so that you can't activate any other debug test without first hitting the ESC key. This way, no matter what uses your keys have during a game, they can't activate debug functions inadvertently.

```

*DEBUG PACKAGE TO SINGLE STEP
      LDA  $C000      ;KEY PRESSED?
      BPL  IGNORE     ;EXIT IF NO KEY PRESSED
      CMP  #$9B       ;ESC KEY?
      BNE  IGNORE
CAUGHT BIT  $C010      ;CLEAR STROBE
      LDA  $C000      ;KEY PRESSED?
      BPL  *-3         ;LOOP BY BRANCHING BACK 3 BYTES
      CMP  #$A0       ;SPACE KEY?
      BNE  IGNORE+3    ;NO,DON'T CLEAR STROBE
IGNORE BIT  $C010      ;CLEAR STROBE
      NOP

```

You could expand the code to do other functions if the code is placed at the block labeled "other tests". Examples of this would be pressing the K key to kill an alien, or the A key to advance to a higher level. This would allow you to reach modules in your code that might take considerable playing time to achieve without your debug module.

Another use for this type of code is to insert a user-controlled pause control into a game. Pause control has just recently been incorporated into arcade games. It is too bad that most programmers hadn't thought of leaving part of the debug module in the game before to offer a pause option.

LASER FIRE & PADDLE BUTTON TRIGGERS

Paddle button switches are used in many games as triggers to fire rockets, bullets and lasers, or to drop bombs. The Apple computer has three; they are numbered 0-2. They are accessed through the addresses \$C061 to \$C063.

To test if a paddle button is pressed, you load the address for that switch into the Accumulator, then test if the value is negative.

```

      LDA  $C061      ;TEST PADDLE #0
      BMI  FIRE       ;NEGATIVE, THEN BUTTON PRESSED
NOFIRE JMP  CONTINUE
FIRE   JSR  LASER     ;FIRE LASER

```

Game designers often want to limit the amount of ammunition that can be fired at one time. A flag can be set to on when a bullet is fired, and to off when the bullet either reaches the opposite end of the screen or if it hits something. The player can't fire again until the flag is in the off position.

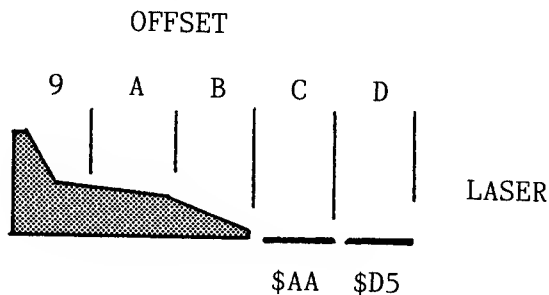
Laser fire presents another problem. The beam travels from the gun or

spaceship to the opposite end of the screen in one frame. If the player held the button, the laser would fire for each frame. Essentially; it would always be on.

The test for a pressed button must include code that would inhibit the button being held down continuously. You can accomplish this by setting a flag to 1 when the laser is fired. If the button is pressed and the laser was just fired without the player releasing it first, the test for the flag prevents it from firing again. The flag is reset to 0 only if the button isn't pressed.

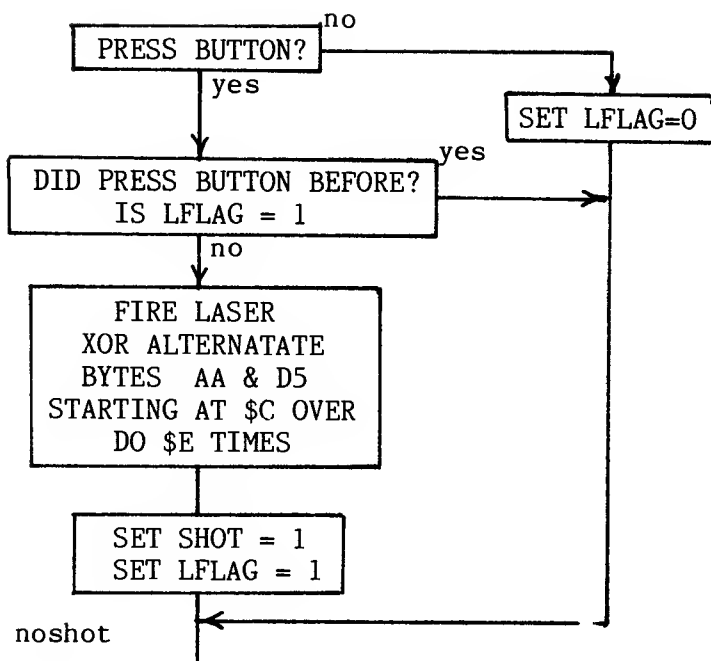
We set another flag called SHOT to one if the laser is fired. This is because we want to XDRAW the laser much later in the animation cycle. If we XDRAW it immediately, it would be barely seen. Yet, if it were automatically XDRAWn later without some sort of test, it would always appear, regardless of whether it was previously fired or not. The XDRAW laser subroutine tests to determine if the SHOT is set before it XDRAWs the laser shot; it will consequently skip this routine if the laser hasn't been fired.

Red lasers look more impressive than white lasers. They also require more work to plot properly. As usual, our nemesis, the even/ odd color offset problem , comes into play. The first position that our laser can be plotted is at horizontal offset \$0C or 12 decimal. This is on an even offset.



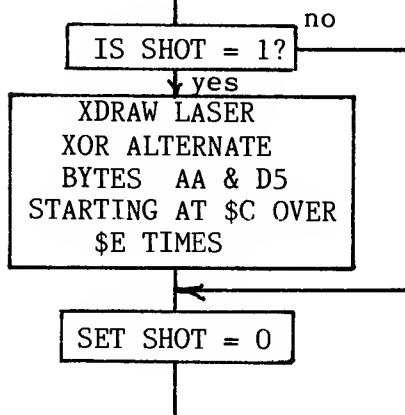
A value of \$AA will produce a red line in even offsets, and a \$D5 will do so in odd offsets. If you plot these two bytes in pairs for \$0E (14 decimal) number of times, you will produce a red laser beam that extends from the plane to the right screen boundary.

A flow chart of our algorithm and its accompanying code follows:



· · ·
· · ·
· · ·

XOR LASER (XLASER)



NOTE: Button has to
be released to reset
LFLAG = 0


```

516 *LASER SUBROUTINE
517 *
63D3: AD 62 CO 518 LASER LDA $C062 ;NEG IF BUTTON PRESSED
63D6: 30 08 519 BMI FIRE1
63D8: A9 00 520 LDA #$00 ;BUTTON NOT PRESSED,SET FLAG TO 0
63DA: 8D 14 60 521 STA LFLAG
63DD: 4C 13 64 522 JMP NOSHOT
63E0: AD 14 60 523 FIRE1 LDA LFLAG ;IS BUTTON BEING HELD DOWN?
63E3: C9 01 524 CMP #$01
63E5: B0 2C 525 BGE NOSHOT
63E7: A9 01 526 LDA #$01
63E9: 8D 13 60 527 STA SHOT ;SET LASER FIRED FLAG
63EC: 8D 14 60 528 STA LFLAG ;SET BUTTON PRESSED FLAG
63EF: 18 529 CLC
63F0: AD 0C 60 530 LDA VERT ;TOP OF SHIP
63F3: 69 07 531 ADC #$07
63F5: A8 532 TAY ;Y REG CONTAINS VERT. LSER POS.
63F6: A9 0C 533 LDA #$0C ;START AT HORIZ=$0C
63F8: 8D 0E 60 534 STA HORIZ
63FB: 20 1C 63 535 JSR GETADR ;FIND ADDRESS OF LASER BEAM LINE
63FE: A2 0E 536 LDX #$0E ;SET UP LOOP FOR E TIMES
6400: A9 AA 537 LASER1 LDA #$AA ;DRAW PAIRS OF AA & D5 BYTES(RED)
6402: 51 26 538 EOR (HIRESL),Y ;BY ORING AGAINST SCREEN
6404: 91 26 539 STA (HIRESL),Y
6406: E6 26 540 INC HIRESL ;NEXT SCREEN POSITION
6408: A9 D5 541 LDA #$D5
640A: 51 26 542 EOR (HIRESL),Y
640C: 91 26 543 STA (HIRESL),Y
640E: E6 26 544 INC HIRESL ;NEXT SCREEN POSITION
6410: CA 545 DEX ;DECREMENT INDEX TO LOOP
6411: D0 ED 546 BNE LASER1 ;DONE?
6413: 60 547 NOSHOT RTS ;YES! EXIT
548 *XDRAW LASER SUBROUTINE
6414: AD 13 60 549 XLASER LDA SHOT
6417: C9 01 550 CMP #$01 ;HAS LASER BEEN SHOT?
6419: D0 24 551 BNE NXSHOT ;NO! SKIP XDRAWING LASER
641B: 18 552 CLC
641C: AD 0C 60 553 LDA VERT
641F: 69 07 554 ADC #$07
6421: A8 555 TAY
6422: A9 0C 556 LDA #$0C
6424: 8D 0E 60 557 STA HORIZ
6427: 20 1C 63 558 JSR GETADR
642A: A2 0E 559 LDX #$0E
642C: A9 AA 560 LASER2 LDA #$AA
642E: 51 26 561 EOR (HIRESL),Y
6430: 91 26 562 STA (HIRESL),Y
6432: E6 26 563 INC HIRESL
6434: A9 D5 564 LDA #$D5
6436: 51 26 565 EOR (HIRESL),Y
6438: 91 26 566 STA (HIRESL),Y
643A: E6 26 567 INC HIRESL
643C: CA 568 DEX
643D: D0 ED 569 BNE LASER2
643F: A9 00 570 NXSHOT LDA #$00 ;RESET LASER FIRED FLAG TO OFF
6441: 8D 13 60 571 STA SHOT
6444: 60 572 RTS

```

COLLISIONS

One of the most important aspects in any arcade game, especially shoot-'em-up type games, is whether an object collides with another object or the background. As a particular object is drawn to the screen, (one byte at a time, or even by single pixels, as some programmers prefer), you can simultaneously test to determine if any other pixels are within that byte's (or pixel's) screen location. The test is performed using the AND instruction.

The truth table for the AND instruction is as follows:

ACC.	MEMORY	RESULT
0	0	0
0	1	0
1	0	0
1	1	1

Both Accumulator and memory must be on (set) for the result to be on (set).

If we take a Hi-Res screen memory location that has an object in it and AND it with a byte from our shape table, any duplication in any bit location because something is already on the screen, will give a non-zero result.

	X	X	X	X		
			X	X	X	X
			X	X		

BACKGROUND

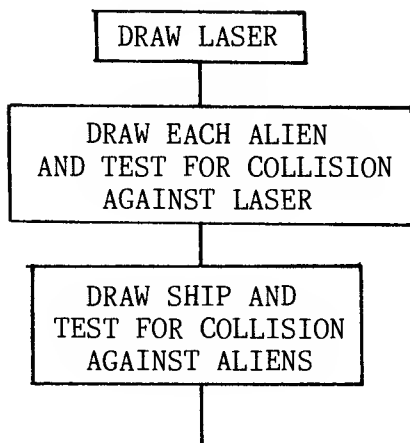
SHAPE

AND BACKGROUND WITH SHAPE
RESULT \$18 > ZERO

The hi bit, (the color control bit), which isn't used to activate any of the seven pixel positions within the byte, could cause a problem. It is possible that if the hi bit were set in an empty or black background (\$80), and a blue or orange shape were ANDed against the screen, the result would be non-zero. Obviously, this is an invalid result, because you can't collide with a black background. The problem can be avoided if the background is first ANDed with #\$7F to mask the hi bit.

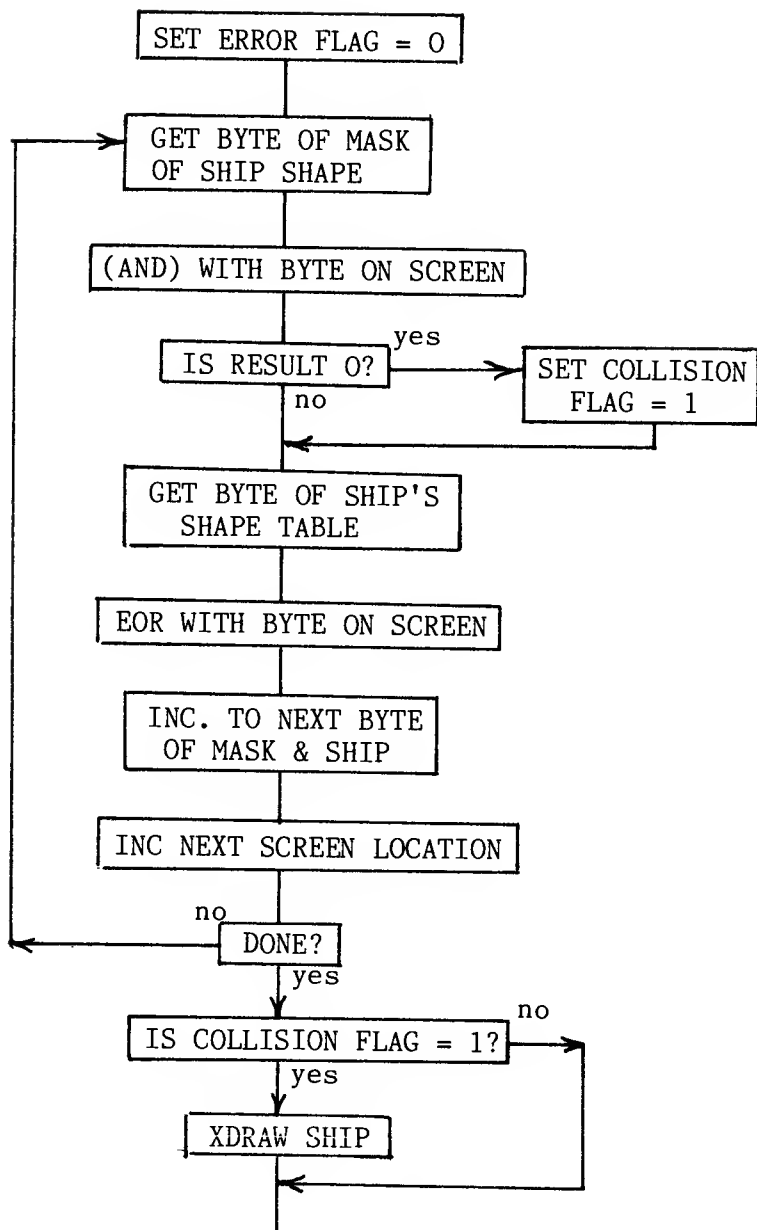
B	O	B	O	B	O	B	HI	
0	0	0	0	0	0	0	1	BACKGROUND
1	1	1	1	1	1	1	0	AND #\$7F
<hr/>								
0	0	0	0	0	0	0	0	RESULT ZERO
0	0	1	0	1	0	1	1	AND BLUE SHAPE
<hr/>								
0	0	0	0	0	0	0	0	RESULT ZERO

The order that each object is drawn is shown in the flow chart below.



There isn't any satisfactory way to avoid the problem of the last test without elaborate testing. Even if we drew the ship first and the aliens last, we wouldn't know if an alien collided with a laser or a ship. It is important that these collision tests be performed before any background, like stars, are drawn to the screen. Also, any permanent background such as ground terrain will always cause a collision.

Single pixel background stars, in some games, are often set in motion to achieve an illusion of speed where stationary ships are involved. Of course, they are drawn and Xdrawn before being moved. Programmers usually keep the star field from intersecting with the ship's range of operation, which usually takes place at the bottom of the screen. However, sometimes it is desirable not to worry about background stars in a program and only draw them at the start of a game. You could adjust the collision counter to ignore single collisions while drawing a complex shape. It is likely that a ship's 24 byte shape would collide with a 16 byte alien shape in more than one place. Small one byte bullets, however, might pose a problem if the collision detector's value were upped to two instead of the usual one.



*DRAW SHIP SUBROUTINE

*DRAW SHAPE ONE LINE AT A TIME-LNGH BYTES ACROSS

*

SDRAW LDA #\$00

STA ESET

SDRAW1 LDY TVERT ;VERTICAL POSITION

JSR GETADR

LDX #\$00

SDRAW2 LDA (STESTL,X) ;GET BYTE OF SHIP MASK SHAPE

AND #\$7F ;MASK OUT HI BIT

AND (HIRESL),Y ;(AND) IT AGAINST SCREEN

CMP #\$00 ; IF ANYTHING IN WAY GET>0

BEQ SDRAW3

LDA #\$01 ;SET BECAUSE IF DON'T FINISH DRAW-

STA ESET ;ING SHIP,PIECE LEFT WHEN XDRAW

*_ ;DURING EXPLOSION

SDRAW3 LDA (SSHPL,X) ;GET BYTE OF SHIP'S SHAPE

EOR (HIRESL),Y

STA (HIRESL),Y ;PLOT

INC STESTL ;NEXT BYTE OF MASK

INC SSHPL ; NEXT BYTE OF TABLE

INY ;NEXT SCREEN POSITION

DEC SLNGH

BNE SDRAW2 ;IF LINE NOT FINISHED BRANCH

INC TVERT ;OTHERWISE NEXT LINE DOWN

DEC DEPTH

BNE SDRAW1 ;DONE DRAWING?

LDA ESET ;IS EXPLOSION FLAG SET?

CMP #\$00

BEQ SDRAW4 ;NO!, EXIT

JMP EXPLODE ;YES!, EXPLODE SHIP

SDRAW4 RTS

EXPLOSIONS

A game wouldn't be complete without the enemy blowing apart when killed. The more dramatic the explosion, the better the effect. Although every programmer has tried it, most have done it the easy way.

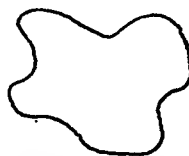
Explosions are divided into two types: shape explosions and particle explosions. Shape explosions are simple, because once an object is targeted for removal, it is replaced first by a garbage-looking shape and then by a white blob, which is larger and resembles a debris-filled fireball.



SHAPE



GARBAGE

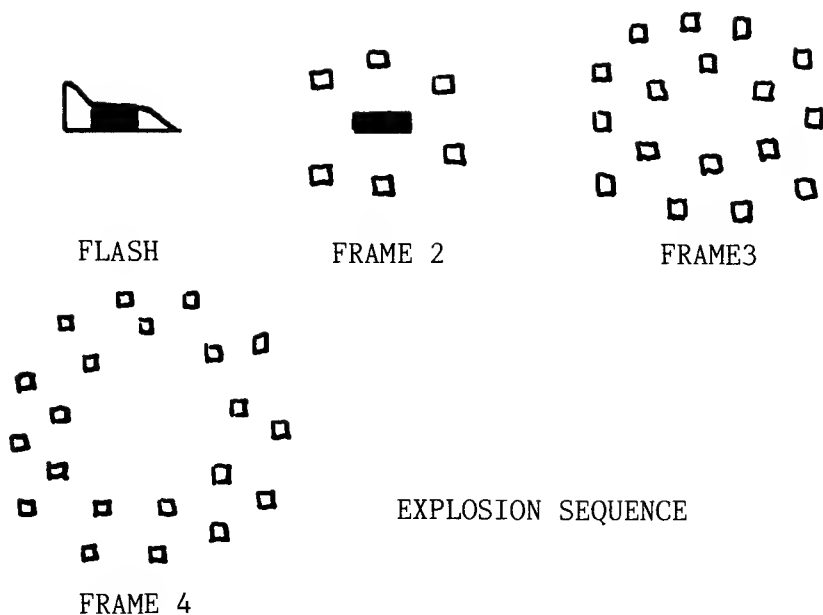


WHITE FIREBALL

The animation is done in successive frames with delays between them. A nice sound routine, which can also act as a delay between plots, is often incorporated. These explosion shapes are stored in a table and are drawn to the screen with drawing subroutines.

Particle explosions are much more complex. They either involve mathematical and random number routines to keep particles streaming outwards from the exploded shape, or they resort to a series of tables to position the particles on the screen. I've chosen the latter case for the following example.

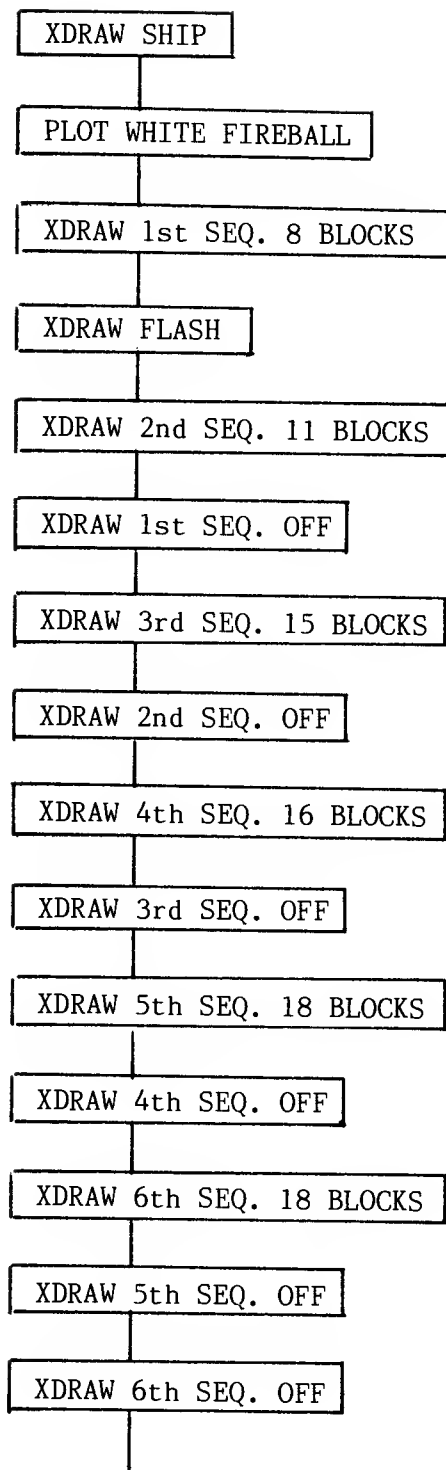
I envisioned a particle fireball that sometimes appears in arcade games like Defender. When the object begins to blow apart, there is a bright flash, then the white hot debris begins expanding in a roughly circular fireball. These fireballs in the arcade grow to be nearly a third the area of the screen and then fade to dull red before blanking out. While fading the particles to red can be included, coding it would be rather difficult. Actually, anything can be done on the Apple if you put your mind to it, but one should weigh the benefits against the time involved. I achieved the basic effect of the explosion in the following manner:

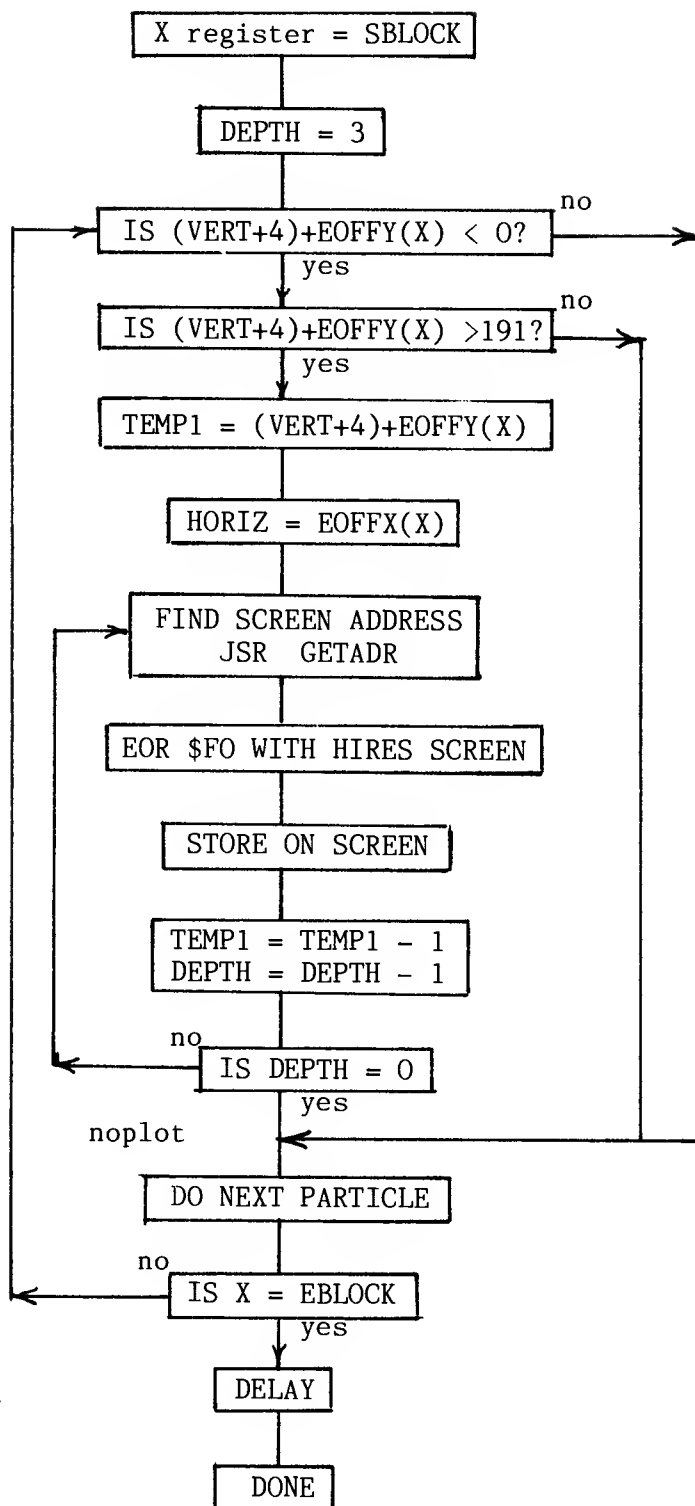


The explosion fills almost 1/9th of the screen. The ship is XDRAWn off the screen and replaced by a bright white block at the ship's center. Then, white particles, each three pixels by four pixels, are drawn in successive expanding but randomized rings. Each frame has a ring of particles, two layers deep. Each successively larger ring requires more particles. The closest ring has only 8 particles, whose positions are stored in two tables, EOFFX and EOFFY. The largest rings have 18 particles.

The two position tables contain the locations of each particle. EOFFX contains the true horizontal offset. EOFFY contains the relative position in relation to the ship's vertical position. For example, the center of the fireball is at VERT + 12. If EOFFY = 8, then the particle is plotted at VERT + 12. And if EOFFY is negative or above the center at -4, it is stored as \$FC (the two's complement), so that it can be added to VERT + 4 directly without testing to see if it is negative, and then subtracting. The number of particles to be plotted in any ring is controlled by SBLOCK and EBLOCK. They determine the start and end points of the data table that is used to draw a ring.

The sequence for drawing the expanding fireball is shown below. It was my choice that only two layers be shown at any one time while the fireball expands. Readers might like to experiment by leaving all of the layers on the screen until the fireball reaches its limit, then XDRAWing them off from the inside out. The time delay in my game may seem fast for most readers. The explosion occurs much too rapidly, but longer delays looked strange using only two layers of debris. Experiment!





```

        667 *EXPLOSION SUBROUTINE
        668 *
6513: 20 1E 65 669 EXPLODE JSR EXPSUB
6516: A9 FE 670 LDA #$FE
6518: 20 A8 FC 671 JSR $FCA8
651B: 4C DA 61 672 JMP FIN
651E: AD OC 60 673 EXPSUB LDA VERT
6521: 8D OD 60 674 STA TVERT
6524: 20 33 63 675 JSR SSETUP ;XDRAW SHIP
6527: 20 FD 62 676 JSR SXDRAW
652A: A9 04 677 EDRAW LDA #$04 ;PLOT WHITE FIREBALL 4 LINES
652C: 8D 11 60 678 STA DEPTH
652F: A9 0A 679 LDA #$0A ;HORIZ POS SHIP'S CENTER
6531: 8D OE 60 680 STA HORIZ
6534: AD OC 60 681 LDA VERT ;VERT POS TOP OF SHIP
6537: 18 682 CLC
6538: 69 04 683 ADC #$04 ;TO REACH CENTER
653A: 8D OD 60 684 STA TVERT
653D: AC OD 60 685 EDRAW1 LDY TVERT ;SHIP'S CENTER
6540: 20 1C 63 686 JSR GETADR
6543: A9 FF 687 LDA #$FF ;WHITE LINE
6545: 51 26 688 EOR (HIRESL),Y
6547: 91 26 689 STA (HIRESL),Y
6549: EE OD 60 690 INC TVERT ;NEXT LINE
654C: CE 11 60 691 DEC DEPTH
654F: D0 EC 692 BNE EDRAW1 ;DONE?
6551: A9 80 693 LDA #$80
6553: 20 A8 FC 694 JSR $FCA8 ;DELAY
        695 *XDRAW SEQ1 -8 BLOCKS
6556: A9 00 696 LDA #$00
6558: 8D 0A 60 697 STA SBLOCK
655B: A9 08 698 LDA #$08
655D: 8D 0B 60 699 STA EBLOCK
6560: 20 1A 66 700 JSR EPLOT
        701 *XDRAW BEGINING FLASH
6563: A9 04 702 EDRAW2 LDA #$04
6565: 8D 11 60 703 STA DEPTH
6568: A9 0A 704 LDA #$0A
656A: 8D OE 60 705 STA HORIZ
656D: 18 706 CLC
656E: AD OC 60 707 LDA VERT
6571: 69 04 708 ADC #$04
6573: 8D OD 60 709 STA TVERT
6576: AC OD 60 710 EDRAW3 LDY TVERT
6579: 20 1C 63 711 JSR GETADR
657C: B1 26 712 LDA (HIRESL),Y
657E: 51 26 713 EOR (HIRESL),Y
6580: 91 26 714 STA (HIRESL),Y
6582: EE OD 60 715 INC TVERT
6585: CE 11 60 716 DEC DEPTH
6588: D0 EC 717 BNE EDRAW3
        718 *XDRAW SEQ2-11BLOCKS
658A: A9 08 719 LDA #$08
658C: 8D 0A 60 720 STA SBLOCK
658F: A9 13 721 LDA #$13
6591: 8D 0B 60 722 STA EBLOCK
6594: 20 1A 66 723 JSR EPLOT
        724 *XDRAW SEQ1- 8 OFF
6597: A9 00 725 LDA #$00

```

6599:	8D	0A	60	726	STA	SBLOCK
659C:	A9	08		727	LDA	#\$08
659E:	8D	0B	60	728	STA	EBLOCK
65A1:	20	1A	66	729	JSR	EPL0T
				730	*XDRAW SEQ3-15	
65A4:	A9	13		731	LDA	#\$13
65A6:	8D	0A	60	732	STA	SBLOCK
65A9:	A9	22		733	LDA	#\$22
65AB:	8D	0B	60	734	STA	EBLOCK
65AE:	20	1A	66	735	JSR	EPL0T
				736	*XDRAW SEQ2-11 OFF	
65B1:	A9	08		737	LDA	#\$08
65B3:	8D	0A	60	738	STA	SBLOCK
65B6:	A9	13		739	LDA	#\$13
65B8:	8D	0B	60	740	STA	EBLOCK
65BB:	20	1A	66	741	JSR	EPL0T
				742	*XDRAW SEQ4-16	
65BE:	A9	22		743	LDA	#\$22
65C0:	8D	0A	60	744	STA	SBLOCK
65C3:	A9	32		745	LDA	#\$32
65C5:	8D	0B	60	746	STA	EBLOCK
65C8:	20	1A	66	747	JSR	EPL0T
				748	*XDRAW SEQ3-15 OFF	
65CB:	A9	13		749	LDA	#\$13
65CD:	8D	0A	60	750	STA	SBLOCK
65D0:	A9	22		751	LDA	#\$22
65D2:	8D	0B	60	752	STA	EBLOCK
65D5:	20	1A	66	753	JSR	EPL0T
				754	*XDRAW SEQ5- 18	
65D8:	A9	32		755	LDA	#\$32
65DA:	8D	0A	60	756	STA	SBLOCK
65DD:	A9	44		757	LDA	#\$44
65DF:	8D	0B	60	758	STA	EBLOCK
65E2:	20	1A	66	759	JSR	EPL0T
				760	*XDRAW SEQ4-16 OFF	
65E5:	A9	22		761	LDA	#\$22
65E7:	8D	0A	60	762	STA	SBLOCK
65EA:	A9	32		763	LDA	#\$32
65EC:	8D	0B	60	764	STA	EBLOCK
65EF:	20	1A	66	765	JSR	EPL0T
				766	*XDRAW SEQ6-18	
65F2:	A9	44		767	LDA	#\$44
65F4:	8D	0A	60	768	STA	SBLOCK
65F7:	A9	56		769	LDA	#\$56
65F9:	8D	0B	60	770	STA	EBLOCK
65FC:	20	1A	66	771	JSR	EPL0T
				772	*XDRAW SEQ5-18 OFF	
65FF:	A9	32		773	LDA	#\$32
6601:	8D	0A	60	774	STA	SBLOCK
6604:	A9	44		775	LDA	#\$44
6606:	8D	0B	60	776	STA	EBLOCK
6609:	20	1A	66	777	JSR	EPL0T
				778	*XDRAW SEQ6-18 OFF	
660C:	A9	44		779	LDA	#\$44
660E:	8D	0A	60	780	STA	SBLOCK
6611:	A9	56		781	LDA	#\$56
6613:	8D	0B	60	782	STA	EBLOCK
6616:	20	1A	66	783	JSR	EPL0T
6619:	60			784	RTS	

```

786 *EXPLOSION PLOTTING SUBROUTINE
787 *
661A: AE OA 60 788 EPLLOT LDX SBLOCK ;LOCATION IN PARTICLE POSITION
789 *- ;TO START DRAWING
661D: A9 03 790 EPLLOT1 LDA #$03 ;EACH BLOCK 3 LINES DEEP
661F: 8D 11 60 791 STA DEPTH
6622: 18 792 ELOOP1 CLC
6623: AD OC 60 793 LDA VERT ;TOP OF SHIP
6626: 69 04 794 ADC #$04 ;NOW CENTER OF SHIP
6628: 18 795 CLC
6629: 7D 9A 69 796 ADC EOFFY,X ;ADD RELATIVE Y POS OF PARTICLE.
662C: C9 00 797 CMP #$00 ;TEST NOT OFF TOP SCREEN
662E: 90 21 798 BLT NOPLOT ;IF OFF, DON'T LOT
6630: C9 C0 799 CMP #$C0 ;TEST NOT OFF BOTTOM SCREEN
6632: B0 1D 800 BGE NOPLOT ;IF OFF, DON'T PLOT
6634: 8D 09 60 801 STA TEMP1 ;STORE VALUE IN TEMP1
6637: BD 44 69 802 LDA EOFFX,X ;LOCATE X POSITION
663A: 8D 0E 60 803 STA HORIZ
663D: AC 09 60 804 ELOOP3 LDY TEMP1 ;FIND LINE ADDRESS TO PLOT ON SCREEN
6640: 20 1C 63 805 JSR GETADR
6643: A9 F0 806 LDA #$F0 ;VALUE OF ALL SHAPE BYTES
6645: 51 26 807 EOR (HIRESL),Y ;XOR WITH SCREEN
6647: 91 26 808 STA (HIRESL),Y ;PLOT ON SCREEN
6649: CE 09 60 809 DEC TEMP1 ;NEXT LINE, IN THIS CASE DRAWING --
664C: CE 11 60 810 DEC DEPTH ;FROM BOTTOM TO TOP
664F: D0 EC 811 BNE ELOOP3 ;DONE?
6651: E8 812 NOPLOT INX ;DO NEXT PARTICLE
6652: EC 0B 60 813 CPX EBLOCK ;DONE WITH ALL PARTICLES IN GROUP?
6655: D0 C6 814 BNE EPLLOT1 ;NO,CONTINUE
6657: A9 30 815 LDA #$30
6659: 20 A8 FC 816 JSR $FCA8 ;DELAY
665C: 60 817 RTS

```

SCOREKEEPING

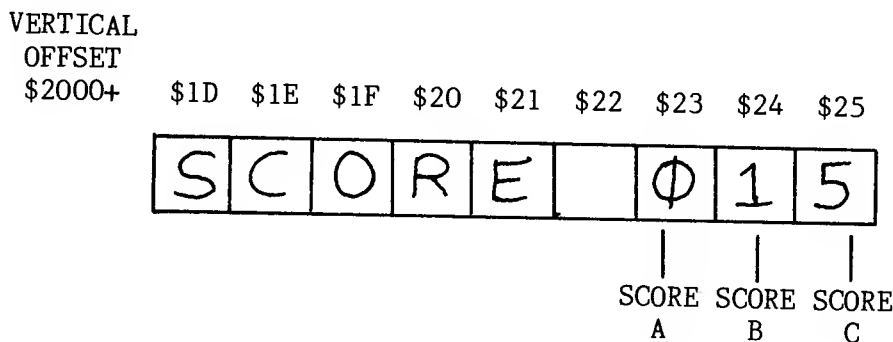
It is a rare exception for machine language games to include a Hi-Res character generator with a complete character set. It is basically a waste of space, because only one or two words are written to the Hi-Res screen along with the numbers 0 through 9 for the numerical score.

For example, in our game, only the word SCORE is written at the top of the screen. This is done once at the start of the game. The numbers, however, change with each alien killed. It would appear that the scoring subroutine would need to convert hexadecimal numbers to decimal numbers, since the computer stores the numerical score as hexadecimal numbers in memory. There is a simple method to avoid this messy approach.

The scoring registers can be broken down into three separate digits, one each for the hundred's digit, ten's digit and one's digit. This is just like the decimal system. Each time an enemy is killed, the one's digit storage location is incremented. This value is tested to see if it becomes greater than 9. If so, the one's digit memory location is reset to zero, and the ten's digit memory location is incremented by one.

If some objects were worth two points instead of one point, we could JSR to SCORE twice. If a target was worth ten points, one could JSR to the middle of the longer SCORE subroutine at a point called SCORE10. This is the place in the subroutine where the ten's digit is incremented. Returning to the main program would be through the usual RTS.

In the following routine, SCOREA represents the one's digit, SCOREB the ten's digit, and SCOREC the hundred's digit. The three variables are drawn on the screen just after the words SCORE, which is on the very first line at the top of the Hi-Res screen.



Since our three digit score doesn't move, the numbers don't change position during the game. Therefore, they don't need to be XDRAWn before being updated. New values can be drawn over the old numbers. This necessitated adding another drawing subroutine that is virtually identical to our standard eight-line deep XDRAW subroutine, but lacks the EOR code. An alternative would be to use your XDRAW drawing subroutine after first blacking out the previous number.

The scoring setup routine is divided into three sections for each of the three digits. SCOREC is to be drawn to the screen at location \$2023, so HIRESL and HIRESH are set appropriately. The ten number shapes which are stored at SCORESH are individually referenced by indexing into a table of 10 byte addresses stored at SCOREP.

```

6A00  SCORESH  HEX 1C 22 .....
6A08                HEX 08 0C .....
6A10                HEX .....

```

```
SCOREP  00 08 10 18 ..
```

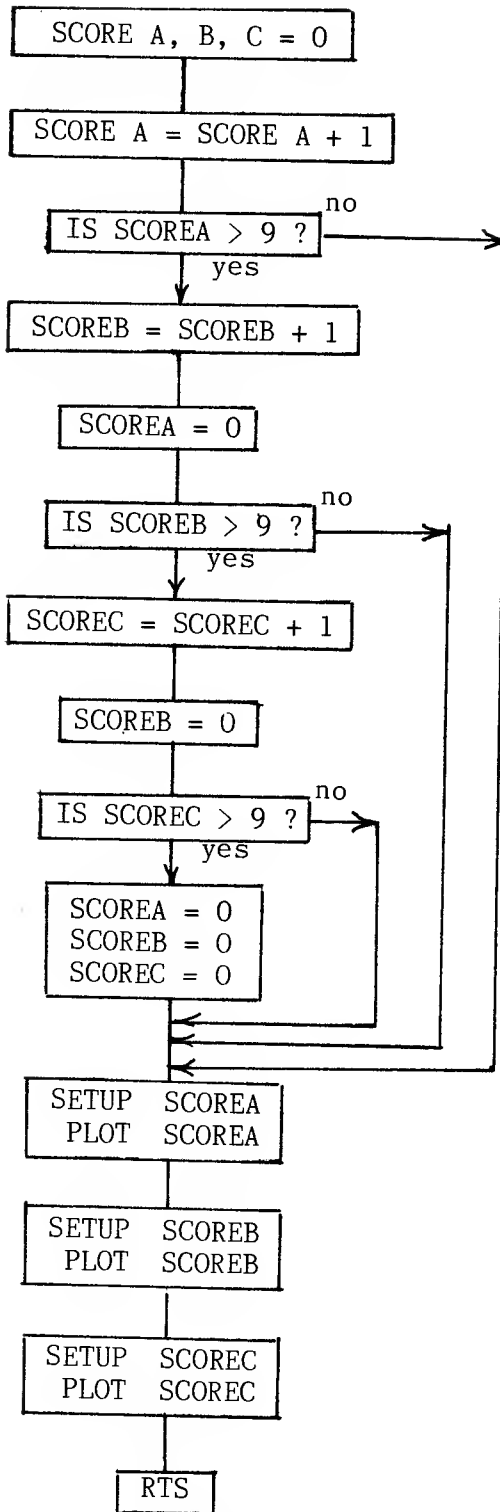
For example, if SCOREC = 2 (hundred's digit), then the Y register contains a 2. LDA SCOREP,Y loads \$10 in the Accumulator and stores the value as SHPL. The hi byte of SCORESH is stored as SHPH. Our drawing routine, using zero page indirect addressing LDA (SHPL),X with X = 0, will reference the correct shape at \$6A10, which in this case are the bytes that form the number 2 on the screen.

The word SCORE stored as a five byte wide, eight-line deep shape, is drawn only once on the screen. This is done at the beginning before the program's main loop.

```

843 *SCORE SETUP ROUTINE FOR DRAW
844 *
6693: A9 20 845 SCRSET LDA #$20
6695: 85 27 846 STA HIRESH
6697: A9 23 847 LDA #$23 ;SETUP SCREEN LOCATION TO PLOT --
6699: 85 26 848 STA HIRESL ;SCOREC ,100'S DIGIT
669B: A9 01 849 LDA #$01 ;DIGIT 1 BYTE WIDE
669D: 8D 10 60 850 STA LNGH
66A0: A9 6A 851 LDA #>SCORESH
66A2: 85 51 852 STA SHPH
66A4: AC 20 60 853 LDY SCOREC
66A7: B9 30 6A 854 LDA SCOREP,Y ;INDEX TO CORRECT SHAPE FOR DIGIT--
66AA: 85 50 855 STA SHPL ;DRAWN
66AC: 20 E8 66 856 JSR SCOREDR ;DRAW 100'S DIGIT
66AF: A9 20 857 LDA #$20 ;SETUP SCREEN LOCATION TO
66B1: 85 27 858 STA HIRESH
66B3: A9 24 859 LDA #$24 ;PLOT SCOREB ,10'S DIGIT
66B5: 85 26 860 STA HIRESL
66B7: A9 01 861 LDA #$01
66B9: 8D 10 60 862 STA LNGH
66BC: A9 6A 863 LDA #>SCORESH
66BE: 85 51 864 STA SHPH
66C0: AC 1F 60 865 LDY SCOREB
66C3: B9 30 6A 866 LDA SCOREP,Y
66C6: 85 50 867 STA SHPL
66C8: 20 E8 66 868 JSR SCOREDR ;DRAW 10'S DIGIT
66CB: A9 20 869 LDA #$20
66CD: 85 27 870 STA HIRESH
66CF: A9 25 871 LDA #$25 ;SETUP SCREEN LOCATION TO
66D1: 85 26 872 STA HIRESL ;PLOT SCOREA, 1'S DIGIT
66D3: A9 01 873 LDA #$01
66D5: 8D 10 60 874 STA LNGH
66D8: A9 6A 875 LDA #>SCORSH
66DA: 85 51 876 STA SHPH
66DC: AC 1E 60 877 LDY SCOREA
66DF: B9 30 6A 878 LDA SCOREP,Y
66E2: 85 50 879 STA SHPL
66E4: 20 E8 66 880 JSR SCOREDR ;DRAW 1'S DIGIT
66E7: 60 881 RTS

```




```

819 *SCORE SUBROUTINE
820 *
665D: EE 1D 60 821 SCORE INC KILLNUM ;ANOTHER ALIEN KILLED
6660: EE 1E 60 822 INC SCOREA ;INCREMENT COUNTER
6663: AD 1E 60 823 LDA SCOREA
6666: C9 0A 824 CMP #$0A
6668: 90 29 825 BLT SCRSET ;IF <10 DON'T CARRY TENS DIGIT
666A: A9 00 826 LDA #$00 ;ZERO OUT 1'S DIGIT
666C: 8D 1E 60 827 STA SCOREA
666F: EE 1F 60 828 SCORE10 INC SCOREB ;ADD CARRY IN TENS
6672: AD 1F 60 829 LDA SCOREB
6675: C9 0A 830 CMP #$0A
6677: 90 1A 831 BLT SCRSET ;IF <10 DON'T CARRY TO 100'S DIGIT
6679: A9 00 832 LDA #$00 ;ZERO OUT 10'S DIGIT & 1'S DIGIT
667B: 8D 1F 60 833 STA SCOREB
667E: EE 20 60 834 INC SCOREC ;ADD CARRY IN 100'S
6681: AD 20 60 835 LDA SCOREC
6684: C9 0A 836 CMP #$0A
6686: 90 0B 837 BLT SCRSET ;SKIP IF LESS 999
6688: A9 00 838 LDA #$00 ;RESET TO 0 IF 1000
668A: 8D 1E 60 839 STA SCOREA
668D: 8D 1F 60 840 STA SCOREB
6690: 8D 20 60 841 STA SCOREC
842 *

```

```

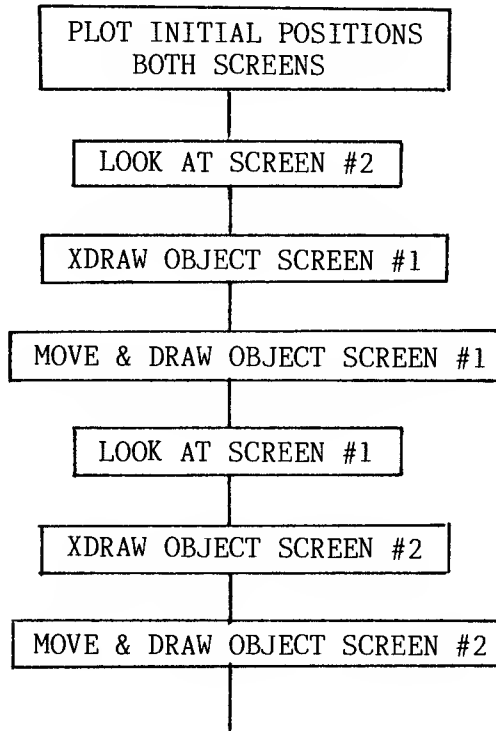
883 *SCORE DRAWING ROUTINE
884 *
66E8: A2 00 885 SCOREDR LDX #$00
66EA: A0 00 886 LDY #$00 ;OFFSET INTO LINE ALREADY SET --
66EC: A1 50 887 SCORED2 LDA (SHPL,X) ;IN SCRSET
66EE: 91 26 888 STA (HIRESL),Y
66F0: A5 27 889 LDA HIRESH
66F2: 18 890 CLC
66F3: 69 04 891 ADC #$04
66F5: 85 27 892 STA HIRESH
66F7: E6 50 893 INC SHPL
66F9: C9 40 894 CMP #$40
66FB: 90 EF 895 BCC SCORED2
66FD: E9 20 896 SBC #$20
66FF: 85 27 897 STA HIRESH
6701: CE 10 60 898 DEC LNCH
6704: F0 03 899 BEQ SCORED3
6706: C8 900 INY
6707: D0 E3 901 BNE SCORED2
6709: 60 902 SCORED3 RTS

```

PAGE FLIPPING

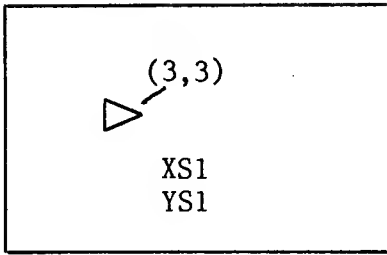
One of the most successful methods for eliminating screen flicker while simultaneously smoothing animation is screen or page flipping. The principle involves drawing on one graphics screen while viewing the other. However, it uses an additional 8K of memory for screen display, and involves elaborate logic to keep track of what and when to draw or erase on a particular screen.

The logic loop for moving an object across the screen is as follows:

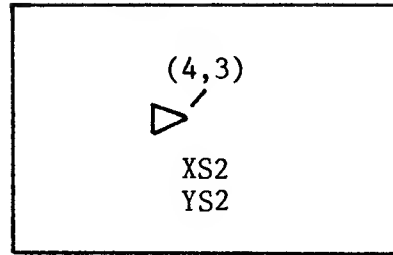


This appears to be rather simple and straight-forward, but it can be tricky. Let's take an object on screen #1, located at X,Y coordinates 3,3. We move it to the right one position to coordinates 4,3 and display it on screen #2. Now, we move it right once more to 5,3 and plot it on screen #1. Before we plot it, we must XDRAW it at its previous position 3,3, because that was its last location on screen #1. This is different from the last location plotted, which is on screen #2. The last time we plotted on screen #1, we plotted our object at 3,3. If you make this mistake and just erase the last object's position, which was actually on the opposite screen, you will XDRAW an object at 3,4 and get an object at that location. Recall that XDRAWing is EORing, and it will plot if nothing is there and erase if something is there.

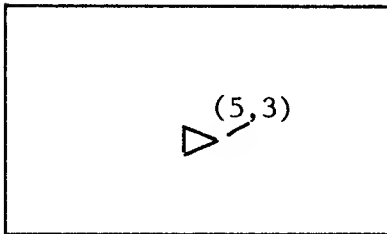
SCREEN #1 PG 1



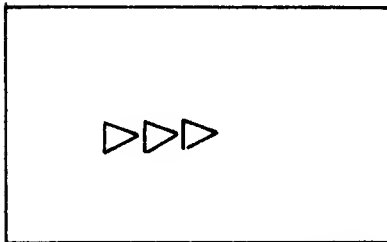
SCREEN #2 PG 2



CYCLE #1



CYCLE #3 CORRECT



CYCLE #3 INCORRECT

Result if XDRAW position of ship Cycle #2 instead of XDRAWing last position on same screen.

The solution to keeping track of the objects is to store the previous location of all objects for both screens. In the above case, XS1,YS1 is always the previous location for the object on screen #1, while XS2,YS2 is the previous screen position for the object on screen #2. While this isn't awkward for one or two objects, a multitude of objects may prove difficult for most programmers. If you are determined to pursue this, I would suggest storing the previous object locations for each screen in tables, which can then be indexed by object number.

To demonstrate a working example of page flipping, the free-floating rocket ship program has been converted to dual screen. Actually, you won't see any

difference in flicker, because only one small object is being drawn. It would require at least a dozen or more objects before you might begin to see the effects of flicker. A small minus sign was added to the bottom left corner of screen #1 as a page reference to determine which screen was being viewed. A single step debug package was also incorporated to allow you to step from screen to screen.

Screen #1 is considered the odd screen and screen #2 the even screen. A counter is incremented for each screen cycle. It is tested for its odd/even character by dividing by two (LSR) and testing the carry bit. Depending on whether COUNTER is odd or even, you might store coordinate values and draw on one screen while displaying the other; then, when COUNTER changes, switch to the opposite screen. For example, if you look at the flow

page flipping DSETUP

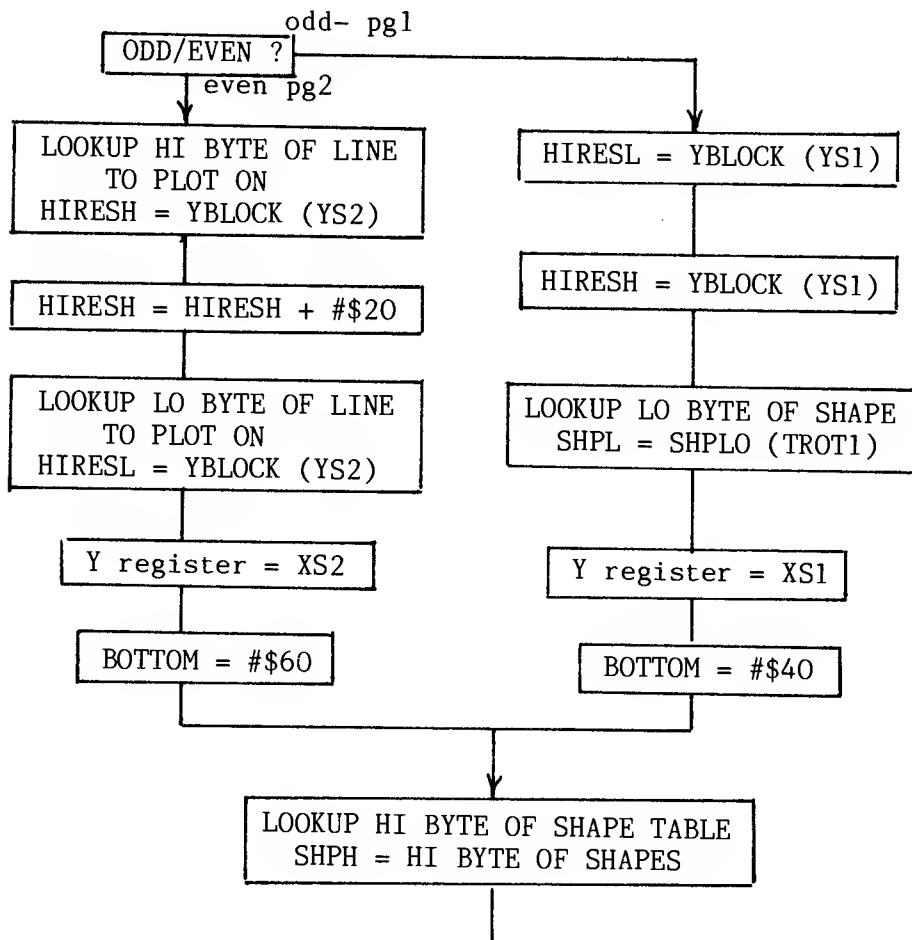
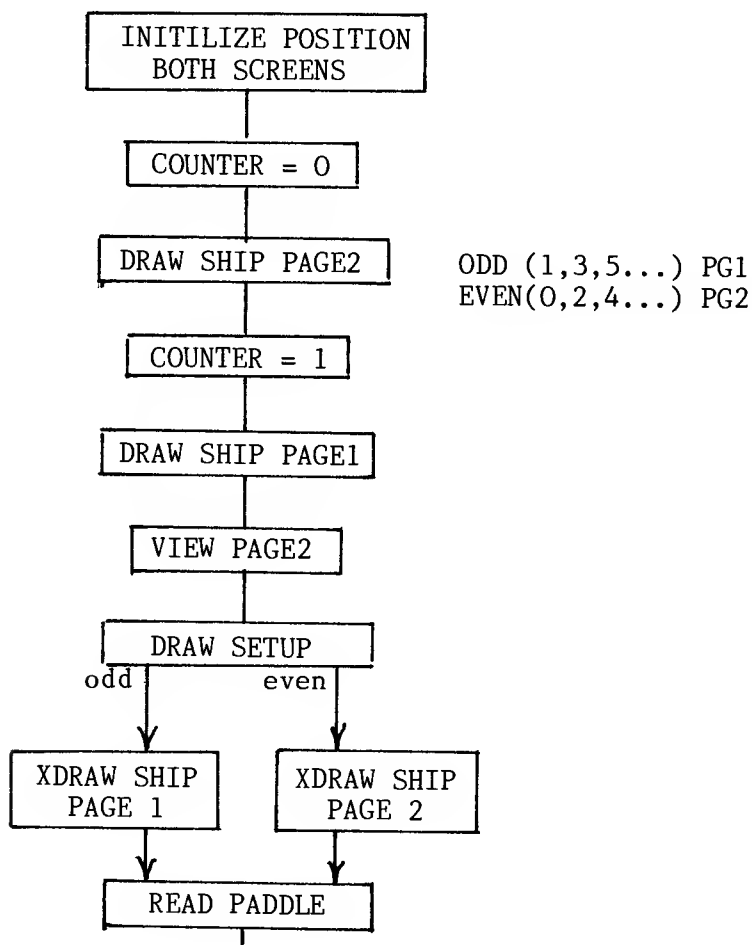
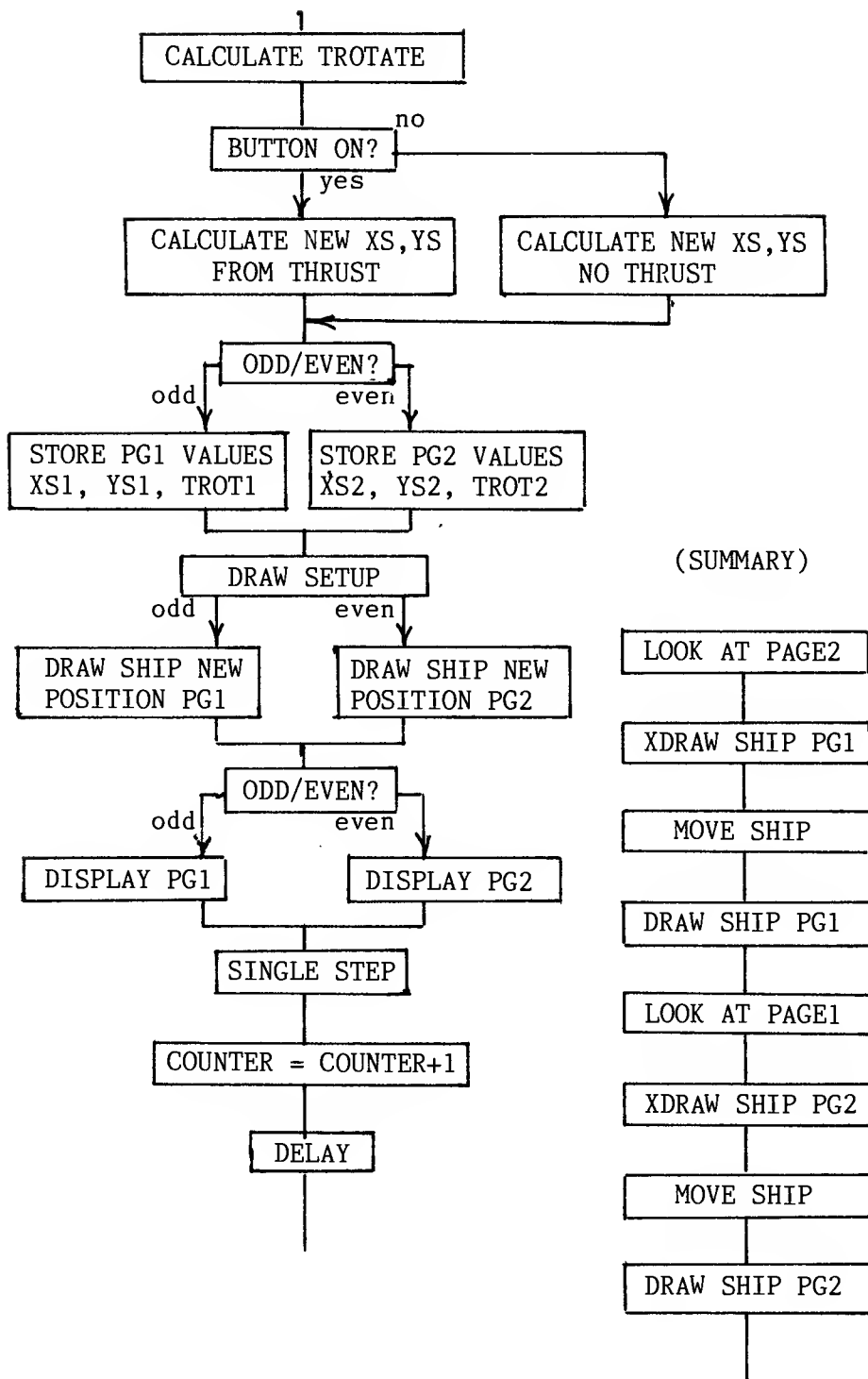


chart below - when COUNTER is even, you store screen #2's values, XS2, YS2, and TROT2 after calculating the ship's new position, and draw the ship on screen #2 while displaying screen #1. When you are finished, you shift the view to screen #2.

Likewise, the drawing setup subroutine must set the pointers to the proper line on the proper screen. An even-valued COUNTER needs to locate the screen line for YS2 and the offset for XS2. In addition, #20 must be added to the hi byte line pointer HIRESH for screen #2. Also, the test to determine if all eight lines have been plotted - a comparison with BOTTOM - becomes \geq #60, which is the end of the second Hi-Res screen.

The flow chart and code is shown below.





```

1  *FREE FLOATING ROCKET (PAGE FLIPPING)
2  ORG $6000
6000: 4C 14 60 3  JMP PROG ;JUMP TO START OF PROGRAM
4  XS DS 1
5  YS DS 1
6  XS1 DS 1
7  XS2 DS 1
8  YS1 DS 1
9  YS2 DS 1
10 VX DS 1
11 VY DS 1
12 PDL DS 1
13 LNCH DS 1
14 COUNTER DS 1
15 BOTTOM DS 1
16 ROTATE DS 1
17 TROTATE DS 1
18 TROT1 DS 1
19 TROT2 DS 2
20 HIRESL EQU $FB
21 HIRESH EQU HIRESL+$1
22 SHPL EQU $FD
23 SHPH EQU SHPL+$1
24 PREAD EQU $FB1E
25 *ENTER HERE FIRST TIME ACCESS
6014: AD 50 CO 26 PROG LDA $C050
6017: AD 52 CO 27 LDA $C052
601A: AD 57 CO 28 LDA $C057
601D: 20 0B 62 29 JSR CLRSCR
6020: 20 25 62 30 JSR CLRSCR2
31 *INITILIZE ROCKET'S STARTING POSITION
6023: A9 14 32 LDA #$14
6025: 8D 03 60 33 STA XS
6028: 8D 05 60 34 STA XS1
602B: 8D 06 60 35 STA XS2
602E: A9 0A 36 LDA #$0A
6030: 8D 04 60 37 STA YS
6033: 8D 07 60 38 STA YS1
6036: 8D 08 60 39 STA YS2
6039: A9 00 40 LDA #$00
603B: 8D 09 60 41 STA VX
603E: 8D 0A 60 42 STA VY
6041: 8D 0F 60 43 STA ROTATE
6044: 8D 11 60 44 STA TROT1
6047: 8D 12 60 45 STA TROT2
604A: A9 00 46 LDA #$00
604C: 8D 0D 60 47 STA COUNTER
604F: 20 BF 61 48 JSR DSETUP ;DRAW EVEN OR PAGE 2 START POS
6052: 20 97 61 49 JSR DRAW
6055: A9 01 50 LDA #$01
6057: 8D 0D 60 51 STA COUNTER
605A: 20 BF 61 52 JSR DSETUP ;DRAW ODD OR PAGE 1 START POS
605D: 20 97 61 53 JSR DRAW
6060: AD 55 CO 54 LDA $C055 ;DISPLAY PG 2 WHILE DRAWING ON PG 1
55 *PUT MINUS SIGN AT BOTTOM LEFT PAGE 2 FOR REFERENCE
6063: A9 FF 56 LDA #$FF
6065: 8D D0 5F 57 STA $5FDO
58 *

```

```

59  ** MAIN PROGRAM LOOP **
60  *
61  * PADDLE READ
6068: 20 BF 61 62 START JSR DSETUP ;WILL SETUP NON DISPLAYED SCREEN
63 *FOR SHIP XDRAW
606B: 20 97 61 64 JSR DRAW ;XDRAW SHIP ON NON DISPLAY SCREEN
606E: A2 01 65 LDX #$01
6070: 20 1E FB 66 JSR PREAD
6073: C0 F9 67 CPY #$F9 ;CLIP VALUE (0-250)
6075: 90 02 68 BLT SKIPP
6077: A0 F8 69 LDY #$F8
6079: 8C 0B 60 70 SKIPP STY PDL
607C: 98 71 TYA
607D: CD 0F 60 72 CMP ROTATE ;PADDLE<ROTATE POS THEN SUBTRACT 5
6080: B0 1B 73 BGE PADDLE3
6082: AD 0F 60 74 LDA ROTATE
6085: 38 75 SEC
6086: E9 05 76 SBC #$05
6088: B0 05 77 BGE PADDLE1 ;MAKE SURE =>0
608A: A9 00 78 LDA #$00
608C: 8D 0F 60 79 STA ROTATE
608F: CD 0B 60 80 PADDLE1 CMP PDL ;DON'T WANT TO GO PAST PADDLE POS
6092: B0 03 81 BGE PADDLE2
6094: AD 0B 60 82 LDA PDL
6097: 8D 0F 60 83 PADDLE2 STA ROTATE
609A: 4C B0 60 84 JMP PADDLE5
609D: CD 0F 60 85 PADDLE3 CMP ROTATE ;PADDLE>ROTATE POS THEN ADD 5
60A0: F0 0B 86 BEQ PADDLE4
60A2: AD 0F 60 87 LDA ROTATE
60A5: 18 88 CLC
60A6: 69 05 89 ADC #$05
60A8: CD 0B 60 90 CMP PDL ;DON'T WANT TO GO PAST PADDLE POS
60AB: 90 03 91 BLT PADDLE5
60AD: AD 0B 60 92 PADDLE4 LDA PDL
60B0: 8D 0F 60 93 PADDLE5 STA ROTATE
60B3: 4A 94 LSR ;DIVIDE BY 16 TO GET ROTATION(0-15)
60B4: 4A 95 LSR ;OR WO ROTATIONS
60B5: 4A 96 LSR
60B6: 4A 97 LSR
60B7: 8D 10 60 98 STA TROTATE
99 *
60BA: AD 62 C0 100 LDA $C062 ;NEG BUTTON PRESSED
60BD: 30 03 101 BMI THRUST
60BF: 4C F7 60 102 JMP NOTHRUST
60C2: AE 10 60 103 THRUST LDX TROTATE
104 *UPDATE VELOCITY VX AND VY
60C5: 18 105 CLC
60C6: BD 6F 62 106 LDA XT,X ;GET X THRUST VECTOR
60C9: 6D 09 60 107 ADC VX
60CC: C9 FD 108 CMP #$FD
60CE: D0 05 109 BNE NOCLIP
60D0: A9 FE 110 LDA #$FE
60D2: 4C DB 60 111 JMP NOCLIP1
60D5: C9 03 112 NOCLIP CMP #$03 ;CLIP MAX VELOCITY AT 2
60D7: D0 02 113 BNE NOCLIP1
60D9: A9 02 114 LDA #$02
60DB: 8D 09 60 115 NOCLIP1 STA VX ;STORE X VELOCITY
60DE: 18 116 CLC
60DF: BD 7F 62 117 LDA YT,X

```


60E2: 6D 0A 60 118		ADC VY	
60E5: C9 FD 119		CMP #\$FD	
60E7: D0 05 120		BNE NOCLIP2	
60E9: A9 FE 121		LDA #\$FE	
60EB: 4C F4 60 122		JMP NOCLIP3	
60EE: C9 03 123	NOCLIP2	CMP #\$03	;CLIP MAX VELOCITY AT 2
60F0: D0 02 124		BNE NOCLIP3	
60F2: A9 02 125		LDA #\$02	
60F4: 8D 0A 60 126	NOCLIP3	STA VY	;STORE Y VELOCITY
		*UPDATE SHIP'S X POSITION XS	
60F7: 18 127	NOTHRUST	CLC	
60F8: AD 09 60 129		LDA VX	
60FB: 6D 03 60 130		ADC XS	
60FE: C9 E0 131		CMP #\$E0	;CHECK FOR WRAPAROUND LEFT
6100: 90 06 132		BLT NWRAP1	
6102: 18 133		CLC	
6103: 69 28 134		ADC #\$28	;FIX BY ADDING 40
6105: 4C 0F 61 135		JMP NWRAP2	
6108: C9 28 136	NWRAP1	CMP #\$28	;CHECK FOR WRAPAROUND RIGHT
610A: 90 03 137		BLT NWRAP2	
610C: 38 138		SEC	
610D: E9 28 139		SBC #\$28	;FIX BY SUBTRACTNG 40
610F: 8D 03 60 140	NWRAP2	STA XS	;STORE SHIP'S NEW X POS
		*UPDATE SHIP'S Y POSITION YS	
6112: 18 141		CLC	
6113: AD 0A 60 143		LDA VY	
6116: 6D 04 60 144		ADC YS	
6119: C9 E0 145		CMP #\$E0	;CHECK FOR WRAPAROUND TOP
611B: 90 06 146		BLT NWRAP3	
611D: 18 147		CLC	
611E: 69 18 148		ADC #\$18	;FIX BY ADDING 24
6120: 4C 2A 61 149		JMP NWRAP4	
6123: C9 18 150	NWRAP3	CMP #\$18	CHECK FOR WRAPAROUND BOTTOM
6125: 90 03 151		BLT NWRAP4	
6127: 38 152		SEC	
6128: E9 18 153		SBC #\$18	; FIX BY SUBTRACTING 24
612A: 8D 04 60 154	NWRAP4	STA YS	; STORE NEW Y POSITION
612D: 18 155		CLC	
612E: AD 0D 60 156		LDA COUNTER	
6131: 4A 157		LSR	
6132: B0 15 158		BCS ODD	
6134: AD 03 60 159	EVEN	LDA XS	
6137: 8D 06 60 160		STA XS2	;STORE SHIP'S CURRENT VARIABLES-PG 2
613A: AD 04 60 161		LDA YS	
613D: 8D 08 60 162		STA YS2	
6140: AD 10 60 163		LDA TROTATE	
6143: 8D 12 60 164		STA TROT2	
6146: 4C 5B 61 165		JMP DONE	
6149: AD 03 60 166	ODD	LDA XS	
614C: 8D 05 60 167		STA XS1	;STORE SHIP'S CURRENT VARIABLES -PG 1
614F: AD 04 60 168		LDA YS	
6152: 8D 07 60 169		STA YS1	
6155: AD 10 60 170		LDA TROTATE	
6158: 8D 11 60 171		STA TROT1	
615B: EA 172	DONE	NOP	
	*		
615C: 20 BF 61 174		JSR DSETUP	;SETUP SHIP'S NEW DRAWING POS
		*FOR NON DISPLAY SCREEN	
615F: 20 97 61 176		JSR DRAW	;DRAW SHIP ON NON DISPLAYED SCREEN
6162: 18 177		CLC	

```

6163: AD OD 60 178      LDA    COUNTER      ;TEST COUNTER TO DETERMINE
                        179  *NEW PAGE DISPLAYE
6166: 4A                180      LSR              ;DISPLAY PAGE JUST DRAWN TO
6167: BO 06            181      BCS    ODD1        ;ODD SHIFT TO PAGE 1
6169: AD 55 CO 182     EVEN1  LDA    $C055        ;EVEN SHIFT TO PAGE 2
616C: 4C 72 61 183      JMP    SKIPO
616F: AD 54 CO 184     ODD1   LDA    $C054
6172: EA                185     SKIPO   NOP
                        186  *DEBUG PACKAGE TO SINGLE STEP
6173: AD 00 CO 187      LDA    $C000        ;KEY PRESSED?
6176: 10 10            188      BPL    IGNORE      ;EXIT IF NO KEY PRESSED
6178: C9 9B            189      CMP    #$9B        ;ESC KEY?
617A: DO OC            190      BNE    IGNORE
617C: 2C 10 CO 191     CAUGHT  BIT    $C010        ;CLEAR STROBE
617F: AD 00 CO 192      LDA    $C000        ;KEY PRESSED?
6182: 10 FB            193      BPL    *-3        ;LOOP BY BRANCHING BACK 3 BYTES
6184: C9 A0            194      CMP    #$A0        ;SPACE KEY?
6186: DO 03            195      BNE    IGNORE+3      ;NO,DON'T CLEAR STROBE
6188: 2C 10 CO 196     IGNORE  BIT    $C010        ;CLEAR STROBE
618B: EA                197      NOP
618C: EE OD 60 198      INC    COUNTER      ;INCREMENT COUNTER FOR NEXT FRAME
618F: A9 CO            199      LDA    $SCO
6191: 20 A8 FC 200      JSR    $FCA8        ; SHORT DELAY
6194: 4C 68 60 201      JMP    START
                        202  *
                        203  * S U B R O U T I N E S *
                        204  *
                        205  *SUBROUTINE TO DRAW ROCKET 1 BYTEBY 8 ROWS
6197: A2 00            206     DRAW   LDX    #$00
6199: A9 01            207      LDA    #$01
619B: 8D OC 60 208      STA    LNGH
619E: A1 FD            209     DRAW2  LDA    (SHPL,X)    ;GET BYTE FROM SHAPE TABLE
61A0: 51 FB            210      EOR    (HIRESL),Y
61A2: 91 FB            211      STA    (HIRESL),Y    ;PUT ON HIRES SCREEN
61A4: A5 FC            212      LDA    HIRESH
61A6: 18                213      CLC
61A7: 69 04            214      ADC    #$04        ;THIS GETS TO NEXT ROW IN BLOCK
61A9: 85 FC            215      STA    HIRESH
61AB: E6 FD            216      INC    SHPL        ;NEXT BYTE OF SHAPE TABLE
61AD: CD OE 60 217      CMP    BOTTOM        ;ARE WE FINISHED WITH 8 ROWS
61B0: 90 EC            218      BCC    DRAW2        ;NO DO NEXT BYTE
61B2: E9 20            219      SBC    #$20        ;RETURN TO TOP ROW
61B4: 85 FC            220      STA    HIRESH
61B6: CE OC 60 221      DEC    LNGH
61B9: FO 03            222      BEQ    DRAW3        ;FINISHED?
61BB: C8                223      INY          ;NEXT COLUMN OF 8 ROWS
61BC: DO EO            224      BNE    DRAW2
61BE: 60                225     DRAW3  RTS
                        226  *DRAWING SETUP SUBROUTINE
61BF: AD OD 60 227     DSETUP  LDA    COUNTER      ;ODD PAGE 1 :EVEN PAGE 2
61C2: 18                228      CLC
61C3: 4A                229      LSR              ;TEST ODD OR EVEN BY SHIFTING -
                        230  *-                ;INTO CARRY BIT
61C4: B0 23            231      BCS    PAGE1
61C6: AC 08 60 232     PAGE2  LDY    YS2
61C9: B9 3F 62 233      LDA    YBLOCKH,Y
61CC: 18                234      CLC
61CD: 69 20            235      ADC    #$20        ;ADD TO REFERENCE SCREEN 2 MEMORY
61CF: 85 FC            236      STA    HIRESH
61D1: B9 57 62 237      LDA    YBLOCKL,Y

```

```

61D4: 85 FB 238 STA HIRESL
61D6: AC 12 60 239 LDY TROT2 ;SETUP POINTER TO CORRECT SHAPE -
240 *- ;TABLE
61D9: B9 8F 62 241 LDA SHPLO,Y
61DC: 85 FD 242 STA SHPL
61DE: A9 60 243 LDA #$60 ;THIS WILL CORRECT DRAWING TEST
244 *FOR END OF 8 LINES - PG 2
61E0: 8D 0E 60 245 STA BOTTOM
61E3: AC 06 60 246 LDY XS2
61E6: 4C 06 62 247 JMP SKIPPY
61E9: AC 07 60 248 PAGE1 LDY YS1
61EC: B9 3F 62 249 LDA YBLOCKH,Y ;LOOK UP HI BYTE OF LINE
61EF: 85 FC 250 STA HIRESH
61F1: B9 57 62 251 LDA YBLOCKL,Y
61F4: 85 FB 252 STA HIRESL
61F6: AC 11 60 253 LDY TROT1
61F9: B9 8F 62 254 LDA SHPLO,Y
61FC: 85 FD 255 STA SHPL
61FE: A9 40 256 LDA #$40
6200: 8D 0E 60 257 STA BOTTOM
6203: AC 05 60 258 LDY XS1 ;DISPLACEMENT INTO LINE
6206: A9 63 259 SKIPPY LDA #>SHAPES
6208: 85 FE 260 STA SHH
620A: 60 261 RTS
262 *CLEAR SCREEN SUBROUTINE
620B: A9 00 263 CLRSCR LDA #$00
620D: 85 FB 264 STA HIRESL
620F: A9 20 265 LDA #$20
6211: 85 FC 266 STA HIRESH
6213: A0 00 267 CLR1 LDY #$00
6215: A9 00 268 LDA #$00
6217: 91 FB 269 CLR2 STA (HIRESL),Y
6219: C8 270 INY
621A: D0 FB 271 BNE CLR2
621C: E6 FC 272 INC HIRESH
621E: A5 FC 273 LDA HIRESH
6220: C9 40 274 CMP #$40
6222: 90 EF 275 BCC CLR1
6224: 60 276 RTS
277 *CLEAR SCREEN 2 SUBROUTINE
6225: A9 00 278 CLRSCR2 LDA #$00
6227: 85 FB 279 STA HIRESL
6229: A9 40 280 LDA #$40
622B: 85 FC 281 STA HIRESH
622D: A0 00 282 CLR3 LDY #$00
622F: A9 00 283 LDA #$00
6231: 91 FB 284 CLR4 STA (HIRESL),Y
6233: C8 285 INY
6234: D0 FB 286 BNE CLR4
6236: E6 FC 287 INC HIRESH
6238: A5 FC 288 LDA HIRESH
623A: C9 60 289 CMP #$60
623C: 90 EF 290 BCC CLR3
623E: 60 291 RTS
292 *TABLES OF STARTING VALUE OF EACH OF 20 BLOCKS
623F: 20 20 21
6242: 21 22 22

```

6245:	23 23 20				
6248:	20	293	YBLOCKH	HEX	20202121222223232020
6249:	21 21 22				
624C:	22 23 23				
624F:	20 20 21				
6252:	21	294		HEX	21212222232320202121
6253:	22 22 23				
6256:	23	295		HEX	22222323
6257:	00 80 00				
625A:	80 00 80				
625D:	00 80 28				
6260:	A8	296	YBLOCKL	HEX	008000800080008028A8
6261:	28 A8 28				
6264:	A8 28 A8				
6267:	50 D0 50				
626A:	D0	297		HEX	28A828A828A850D050D0
626B:	50 D0 50				
626E:	D0	298		HEX	50D050D0
		299	*		
626F:	00 01 01				
6272:	01 00 FF				
6275:	FF FF	300	XT	HEX	0001010100FFFFFF
6277:	00 01 01				
627A:	01 00 FF				
627D:	FF FF	301		HEX	0001010100FFFFFF
627F:	FF FF 00				
6282:	01 01 01				
6285:	00 FF	302	YT	HEX	FFFF0001010100FF
6287:	FF FF 00				
628A:	01 01 01				
628D:	00 FF	303		HEX	FFFF0001010100FF
		304	*		
628F:	03	305	SHPLO	DFB	SHAPES
6290:	0B	306		DFB	SHAPES+\$08
6291:	13	307		DFB	SHAPES+\$10
6292:	1B	308		DFB	SHAPES+\$18
6293:	23	309		DFB	SHAPES+\$20
6294:	2B	310		DFB	SHAPES+\$28
6295:	33	311		DFB	SHAPES+\$30
6296:	3B	312		DFB	SHAPES+\$38
		313	*NEXT GROUP BECAUSE PADDLE (0-15) INDEXES INTO		
		314	*SHAPE TABLE TWICE		
6297:	03	315		DFB	SHAPES
6298:	0B	316		DFB	SHAPES+\$08
6299:	13	317		DFB	SHAPES+\$10
629A:	1B	318		DFB	SHAPES+\$18
629B:	23	319		DFB	SHAPES+\$20
629C:	2B	320		DFB	SHAPES+\$28
629D:	33	321		DFB	SHAPES+\$30
629E:	3B	322		DFB	SHAPES+\$38
		323	*		
		324	SPACE	DS	100
		325	*ROCKET	SHAPES	
6303:	00 08 08				
6306:	08 1C 1C				
6309:	36 00	326	SHAPES	HEX	000808081C1C3600
		327	*2ND		

```

630B: 00 00 20
630E: 14 0F 1C
6311: 08 08      328      HEX  000020140F1C0808
           329      *3RD
6313: 00 00 02
6316: 0E 7C 0E
6319: 02 00      330      HEX  0000020E7C0E0200
           331      *4TH
631B: 00 08 08
631E: 1C 0F 14
6321: 20 00      332      HEX  0008081C0F142000
           333      *5TH
6323: 00 00 36
6326: 1C 1C 08
6329: 08 08      334      HEX  0000361C1C080808
           335      *6TH
632B: 00 08 08
632E: 1C 78 14
6331: 02 00      336      HEX  0008081C78140200
           337      *7TH
6333: 00 00 20
6336: 38 1F 38
6339: 20 00      338      HEX  000020381F382000
           339      *8TH
633B: 00 00 02
633E: 14 78 1C
6341: 08 08      340      HEX  00000214781C0808

```

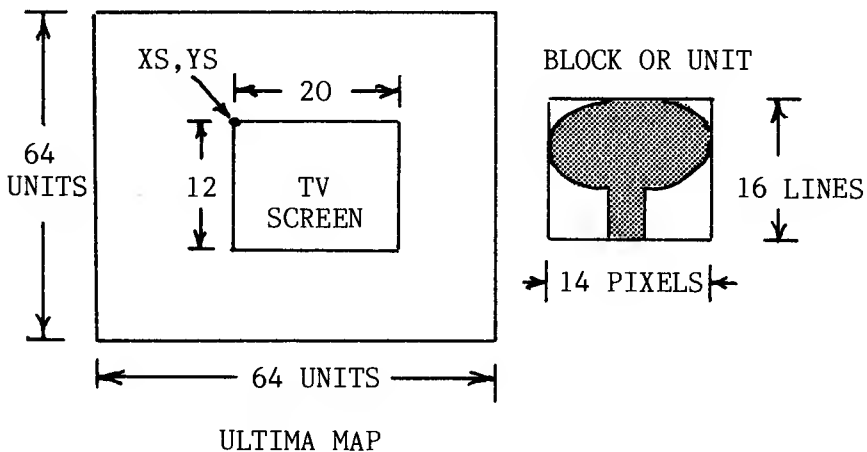
--END ASSEMBLY--

ERRORS: 0

835 BYTES

GAMES THAT SCROLL

Scrolling games are dynamic in nature, in that the entire background moves as the player traverses the game's terrain. True scrolling arcade games, such as Pegasus II on the Apple, or Scramble and Rally X in the arcades, have multi-screen worlds which scroll on or off the screen as the player's plane or car moves. These games show only a window or part of the entire background world at one time. They differ from games that have background stars and aliens that appear to be traveling towards you from top to bottom. Scrolling games have objects or terrain in relatively stable positions within the game's world. They can be reached by traveling to that particular section of the world. And this technique isn't just limited to arcade games. Ultima, an adventure game, uses a large map that scrolls as the player moves around. Your screen view is only a small window on the game's world.

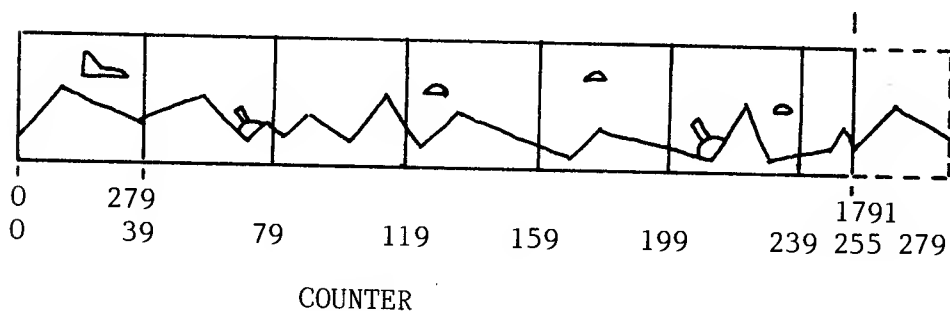


The data that generates these maps is stored in large arrays. A game like Ultima has a map 64 units square, with each block 14 pixels wide by 16 lines deep. If one byte is used to store which shape is used for each block, 4K of memory is needed. There is a reason why 64 units was chosen for a side. When referencing the location of your viewing window, which is located at position XS, YS on the large map, you retrieve data from a table or array, in which each row of blocks is stored \$40 below the previous row. Sixty-four units per side is not etched in concrete, but some multiple of 16 is convenient. A map 128 units by 32 units would also work well.

Games like Pegasus II on the Apple allow as many as ten screen lengths to scroll past the viewer before repeating. The horizontal scrolling is done a byte at a time, and the data is stored in tables. Pegasus II, which uses page flipping to smooth the animation, gains added speed by scrolling only sections of the screen.

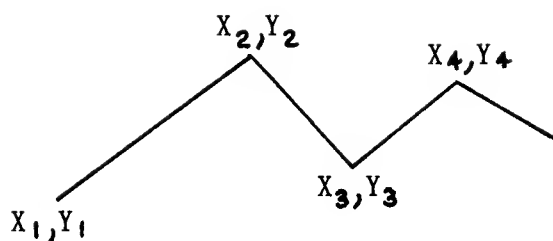
In this section, we are going to develop a scrolling game much like Pegasus II. It will be defined in much more detail than my previous examples, yet it won't be complete. Aliens will appear, but they won't shoot back. You'll be able to kill the aliens with your lasers and accumulate points as you do so, but you'll find that there is no finish, nor even a goal. Consider the unfinished game a test bench to develop your graphics skills.

The first step is to define and develop a fast scrolling subroutine. Since it is easier to move objects horizontally one byte per animation frame, our scrolling should be linked with that speed if objects are to remain synchronized with the terrain. A counter can be used to determine the screen's location within our much larger world. With the counter limited to 256 and screen scrolling set at 7 pixels per frame, the most logical length for a world would be 1792 pixels or seven screen lengths.



When the counter reaches 256, it wraps back to zero for a repeat of screen #1. You have to be careful when approaching the upper end of the database. Once the counter indexes beyond 215, it begins accessing data beyond the 1791st position. This can be remedied by enlarging the table to 2048 data points, with the last 279 points a duplicate of the first 279 points. The terrain level at the end of the seventh screen should match the terrain level at the beginning of the first frame, as shown above.

The data points are Y axis screen coordinates (0-191) for each of the 1792 positions along the X axis. The data was placed into the table by an Applesoft program called Mountain Maker. It takes a series of X,Y points corresponding to each change in direction of our terrain and, by simple slope equations, generates the data points in between. The program is listed below.



$$\text{SLOPE} = \frac{\Delta Y}{\Delta X} = \frac{Y_2 - Y_1}{X_2 - X_1} = \frac{Y - Y_1}{X - X_1}$$

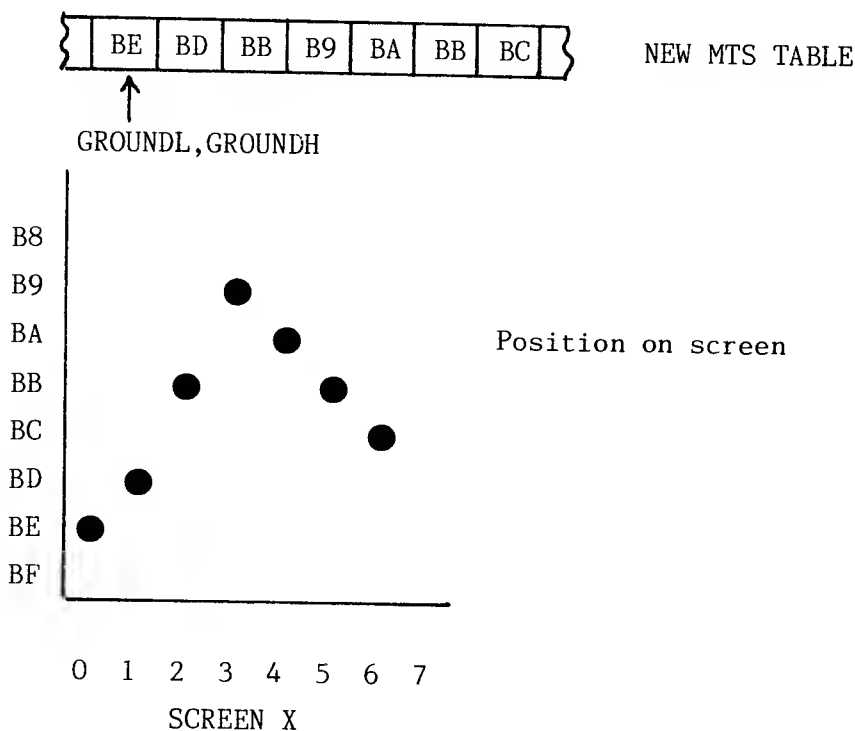
$$Y = Y_1 + \left[\left(\frac{Y_2 - Y_1}{X_2 - X_1} \right) (X - X_1) \right]$$

```

5  DIM NAME$(20)
10  TEXT : HOME : PRINT : PRINT "    MOUNTAIN BACKGROUND GENE
RATOR"
20  PRINT : HTAB 15: PRINT "WORKING"
25  SH = 4000
30  START = 16384
35  J = START
40  READ A,B
50  X2 = A:Y2 = B
60  READ C,D
70  IF C = - 1 THEN 1000
80  X1 = X2:Y1 = Y2:X2 = C:Y2 = D
90  SLOPE = (Y2 - Y1) / (X2 - X1)
100 FOR I = X1 TO X2 - 1
105 Y = INT (Y1 + (SLOPE * (I - X1)))
110 POKE J,Y
120 J = J + 1
130 NEXT I: GOTO 60
150 END
1000 POKE J,Y2
1010 PRINT : INPUT "DATABASE NAME ?";NAME$
1020 PRINT "BSAVE";NAME$;" ,A$";SH;" ,L$2000"
2000 DATA 0,10,80,40,175,25,250,65,335,20,375,32
2010 DATA 625,32,700,15,750,70,900,45,1070,90
2020 DATA 1190,12,1220,20,1320,10,1350,17,1440,5
2030 DATA 1500,40,1540,100,1610,50,1640,40,1710,5
2040 DATA 1730,5,1810,15,1840,15,1870,35,1900,25,1920,55,19
50,30,1980,55
2050 DATA 2047,10,-1,-1

```


The scrolling subroutine works as follows. Each time the position counter, INDEX, is incremented, it adds seven to the lo byte of a pair of zero page pointers, GROUNDL and GROUNDH, through a multi-byte addition. These pointers index into a table called NEW MOUNTAINS, stored at \$4000. Starting with the first data point located at GROUNDH, GROUNDL, the routine plots that point at $X = 0$. It increments the lo byte of the data point, then plots the second point at $X = 1$. It does that until all 280 points are plotted. Plotting is accomplished by EORing the proper pixel to the screen. When it is finished plotting, it reloads GROUNDH and GROUNDL, then EORs all the points off the screen. Note that GROUNDH and GROUNDL are not changed during the plotting phase because zero page locations \$4 and \$5 were used to store the pointers. When these are incremented, it doesn't affect our original pointers, which are stored elsewhere.



The terrain does flicker excessively because it is off the screen as much as on the screen. I'm sure ambitious readers will want to rewrite the subroutine, or convert the entire program to page flipping.

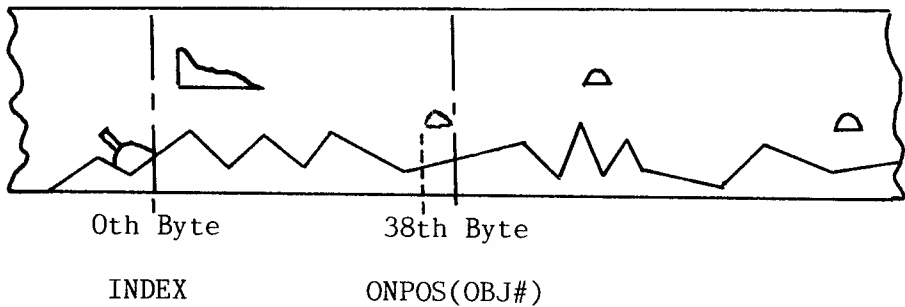
The second step in developing the game is to devise a method for determining whether an object is on or off the screen. This depends on the location of the object in our multi-screen long world in relation to that of the screen's moving window. Obviously, the two must coincide for the object to appear.

Our viewing window is controlled by the counter, INDEX (0-255). We see the terrain in that window from INDEX * 7 to (INDEX + 39) * 7. While our terrain is stored as individual data points for each pixel, our shapes are stored and plotted as data bytes at a particular horizontal position (0-39).

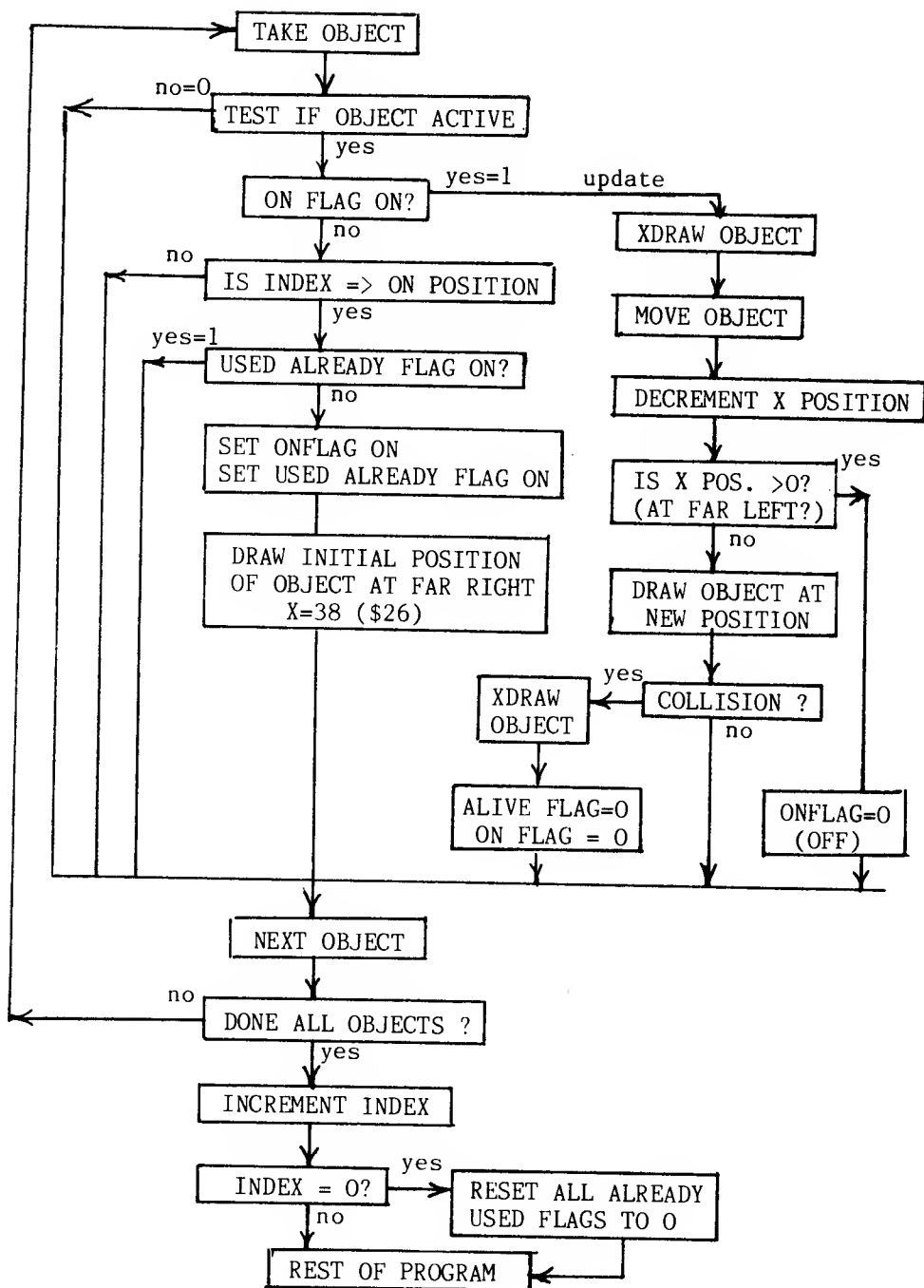
Fortunately, the choice of moving the terrain seven pixels (or one screen byte to the left with each frame) synchronizes with the easiest method of moving a raster shape in the same direction. Single byte moves require no offset shape tables.

Objects can be assigned reference positions corresponding to their horizontal byte location (0-255) in our seven screen long world. A table of these values is stored in ONPOS. Each object's vertical position is correspondingly stored in a table TABLEY. TABLEX contains the object's current screen position (0-39). This value changes during each frame, regardless of whether the object remains stationary with respect to the terrain.

An object first appears on the scrolling screen at the far right when INDEX = > ONPOS(OBJ #). The ONPOS value for an object is not actually its true horizontal position, but one that is offset by 39 bytes.



The object moves left one byte exactly in step with the ground movement with each successive animation frame. The value of TABLEX (OBJ #) is set originally to X = 38 or \$26. X is set to 38 rather than 39 because our alien shape is two bytes wide, and we would like to plot its full shape on the screen's right side rather than half of its shape. During each successive cycle, we decrement the X position in TABLEX table and test each time for a value less than zero. If so, we are now off the screen, and we set the ONFLAG (OBJ #) = 0



There are several flags that are required to keep track of certain aspects of the game. The ONFLAG (OBJ #) is used to determine if the object is to be actively plotted on the screen. Assuming our object is actually alive, ALIVE (OBJ #) = 1 and not dead (value = 0), then the ONFLAG (OBJ #) is tested. If this flag was turned on because the object meets the INDEX = > ONPOS (OBJ #) test, it will appear for the next 38 cycles unless it is destroyed by your ship's laser. In either case, when the object reaches the end of its time on the screen, the ONFLAG (OBJ #) flag is set to off, or zero.

There is one additional flag. That is the USFLAG, or used-already flag. It is necessary because if, for example, an object were to appear on the screen when INDEX = 50 and vanish at INDEX = 88, without this flag being set equal to one (off), the object would again meet the requirements of INDEX = > ONPOS (OBJ #) as soon as the ONFLAG (OBJ #) was zero. The object would appear every 38 screen cycles after it first appeared until INDEX wrapped around to become zero again. The object should appear only once over the (0-255) INDEX cycle. Incidentally, once all objects have been tested and plotted and INDEX = 0 again, the program resets all USFLAG (OBJ #) = 0 so that they will reappear over the same terrain if they are still alive.

Collisions are tested during the draw routine. The collision flag, KILL, is set if any lit pixel occupies the screen positions, where an alien or saucer shape is drawn. The test is made by logically ANDing the shape with the screen. A non-zero value will set the flag. If a collision is detected, the alien is immediately XDRAWn off the screen, and both the ALIVE flag and the ONFLAG are set to zero (off) for that object. Of course, in a real game, you wouldn't have an alien simply disappear, but would either plot the shape of an explosion or blow it up dramatically; a fitting end that any alien who travels so far and fights so valiantly deserves.

I'll admit that the routine is quite complex and did require considerable planning and thought, but I hope that the accompanying flow chart will make it clear. Remember that this code is looped for each object successively until all objects are tested. Only then does it increment INDEX before proceeding on with the rest of the program.

Flexibility for displaying a variety and a large number of shapes, plus the ability to change the placement of these shapes, was designed into the program. This becomes extremely helpful during the play test when the quantity of targets and types are liable to change frequently. Ground based laser, radar and rocket bases, plus a dozen city buildings were envisioned as targets spread out over seven screens. While only eight different shapes were contemplated, ten of one type might be needed, while only three of another type might be used.

Because of this special need, a table called SHPADR was conceived. It would hold the shape type for each, and as many as 256 targets. The shapes would be stored in a shape table called SHAPES. Since each shape was two bytes wide by eight lines deep, and we need both even and odd offset shape tables for color, thirty two bytes would be required for each shape. To keep the

table within one page boundary (256 bytes), the scheme was limited to eight shapes.

SHAPES

SHAPE #0	EVEN
SHAPE #1	EVEN

. .
. .

SHAPE #7	EVEN
SHAPE #0	ODD

. .
. .

THE 8 ODD OFFSET SHAPES
FOLLOW THE 8 EVEN OFFSET
SHAPES IN THE TABLE
CALLED SHAPES.

Another table, called SHPLO, is used to reference the lo byte of each shape. The values in this table are permanently set, starting at \$00 and increasing by \$10 with each shape. However, because we are using only two shapes in this example, and loading the shape table after assembling is an extra step, it is easier during program development to have the assembler construct the table for us by using the DFB pseudo-op code to define the lo order byte.

Thus, the SHPLO table is constructed as follows for the two shapes:

```

SHPLO  DFB SHAPES      ;LO BYTE ALIEN  EVEN OFFSET
        DFB SHAPES+$10 ;LO BYTE SAUCER EVEN OFFSET
        DFB SHAPES+$20 ;LO BYTE ALIEN  ODD  OFFSET
        DFB SHAPES+$30 ;LO BYTE SAUCER ODD  OFFSET

```

The table SHPADR for seven objects either points to shape #0 (alien) or shape #1 (saucer). It actually indexes into SHPLO to set the proper pointers.

```

EVEN  LDY  SHPADR,X ;WHERE X IS THE OBJECT #
      LDA  SHPLO,Y  ;PROPER LO BYTE OF EVEN OFFSET SHAPE
      STA  SHPL

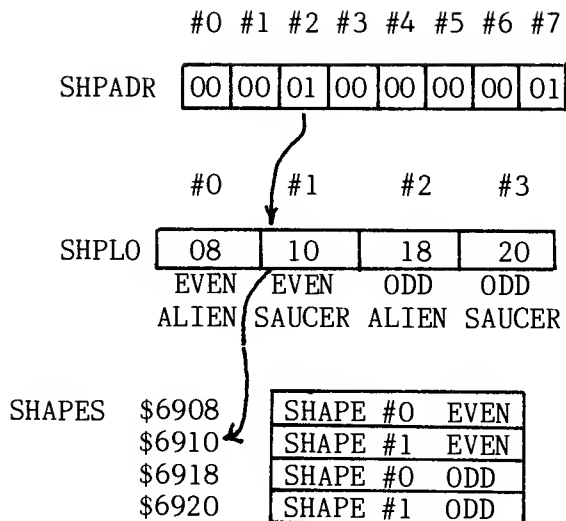
```

The code for the odd offset is similar, except you have to index into the odd half of SHPLO which, in this case, begins with the third byte.

```

ODD   LDY  SHPADR,X
      LDA  SHPLO+2,Y ;PROPER LO BYTE OF ODD OFFSET SHAPE
      STA  SHPL

```



For example, if you were to look for object #2 (X reg = 2), which is an even number, the even code would reference \$01 for the SHPADR table. This in turn would point to the #1 element in SHPLO. Thus, the code would be stored \$10 in SHPL. The high byte \$69 would be stored in SHPH.

In the event that you chose to place these tables into a permanent location, skip the construction of the SHPLO table. Instead, the SHPADR table contains the lo byte for each shape. The SHPADR table's length is doubled, for it now contains the locations of both the even and odd shapes.

SHAPES	\$7000	SHAPE #0	EVEN
	\$7008	SHAPE #1	EVEN
	\$7010	SHAPE #0	ODD
	\$7018	SHAPE #1	ODD

	#0	#1	#2	#3	#4	#5	#6	#7
SHPADR	00	00	08	00	00	00	00	08
	10	10	18	10	10	10	10	18

The corresponding code is as follows:

```

EVEN  LDY  SHADR,X
      STA  SHPL

ODD   LDY  SHPADR+8,X
      STA  SHPL

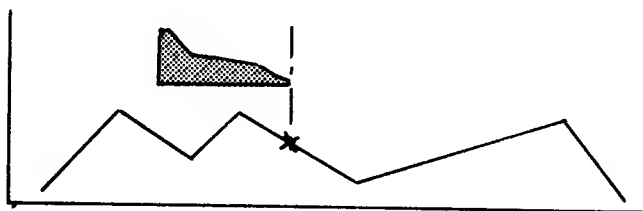
```

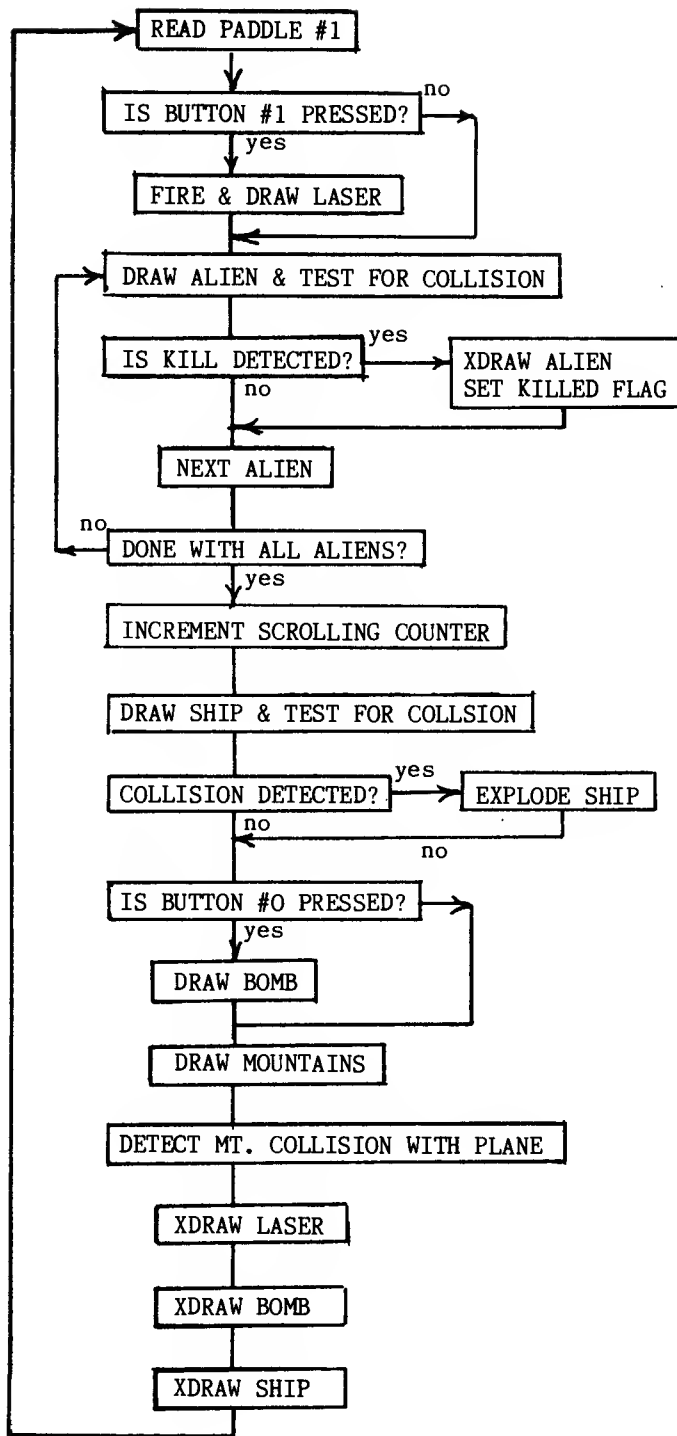
You can see that this is actually simpler code. If you wish to keep separate shape tables independent of the main program's code, then this is the preferred method. However, it does involve loading your shape table into memory when testing a program.

ORDER OF EVENTS IN GAME

The sequence of events in any game is important. Sometimes the order is dictated by tests performed by various routines. It becomes obvious that you can't test for a collision of an alien with a laser beam unless the laser is drawn on the screen first. You can't determine if your ship collides with an alien unless the ship is drawn last. Unfortunately, something is always last. A collision of the ship with an alien at this point in the sequence requires testing each alien's screen coordinates to determine which one hit the ship.

The mountains were drawn afterwards to minimize the objects' screen flicker. Since the mountain routine takes considerably longer to draw than the rest of the objects combined, it acts as a time delay, allowing the objects to remain on the screen longer than they are off. Because the mountains are drawn after the ship's collision test, a separate test was devised for mountain collisions. The code compares the ship's vertical position with the vertical value of the mountain data drawn directly beneath it. The ship's vertical position must be less than the value referenced in the mountain data table (i.e., ship is above mountains). Remember that MTOFFL and MTOFFH points to the beginning position in the table from which the scroll subroutine draws the next 280 points of the mountain background. The tip of the ship is located at $X = 84$ or $\$55$. The collision test is at the nose, so $\$55$ is added to MTOFFL. Since the carry is not cleared when $\$55$ is added to the offset location of the mountain table, an overflow in the lo byte, which is a carry set, automatically increments the hi byte value. Both the lo and hi byte values are stored at $\$09$ and $\$0A$, respectively, in the zero page. These were chosen as scratch memory locations in zero page to do an indirect indexed load, $(LDA (\$09),Y)$, where the Y register is zero. This obtains the value of the mountain pixel directly below the ship's nose, and with only one instruction! This is compared with the vertical position of the ship's bottom. If the value in the mountain table is greater, there is no collision.

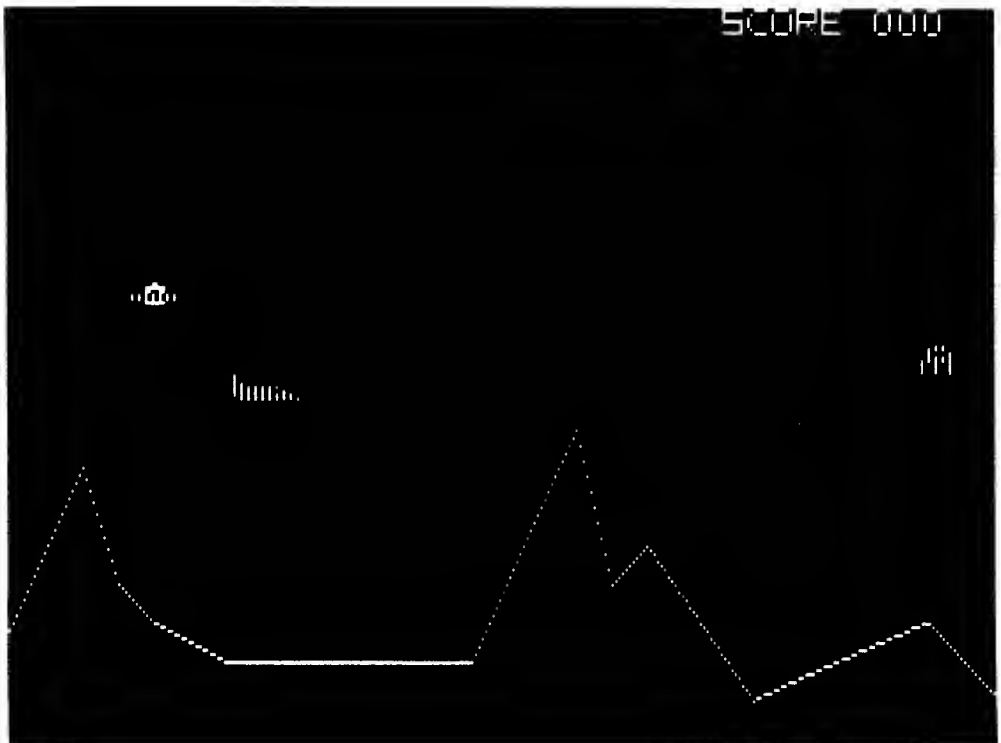





```

211  *DETECT FOR MT COLLISION
212      LDA  PADDLEL
213      CLC
214      ADC  #$55          ;TIP OF SHIP @84
215      STA  $09
216      LDA  PADDLEH
217      ADC  #$40          ;LOCATION OF MOUNTAIN TABLE
218      STA  $0A
219      LDY  #$00
220      CLC
221      LDA  VERT
222      ADC  #$08          ;BECAUSE PDL IS AT TOP OF PLANE--
223      STA  TEMP          ;AND MOUNTAINS HIT BOTTOM
224      LDA  ($09),Y
225      CMP  TEMP
226      BGE  NOHIT
227      JMP  EXPLODE
228 NOHIT  LDA  VERT

```



```

1      *COMPLETE SCROLLING GAME CODE
2      ORG $6000
6000: 4C 21 60 3      JMP PROG          ;JUMP TO START OF CODE
4      COUNT DS 1
5      INDEX DS 1
6      PADDLEL DS 1
7      PADDLEH DS 1
8      PDL DS 1
9      TEMP DS 1
10     TEMP1 DS 1
11     SBLOCK DS 1
12     EBLOCK DS 1
13     VERT DS 1
14     TVERT DS 1
15     HORIZ DS 1
16     OBJ DS 1
17     LNGL DS 1
18     DEPTH DS 1
19     SLNGH DS 1
20     SHOT DS 1
21     LFLAG DS 1
22     ESET DS 1
23     BVERT DS 1
24     TBVERT DS 1
25     BVELY DS 1
26     BHORIZ DS 1
27     BMLOCK DS 1
28     TBMLOCK DS 1
29     KILL DS 1
30     KILLNUM DS 1
31     SCOREA DS 1
32     SCOREB DS 1
33     SCOREC DS 1
34     HIRESL EQU $26
35     HIRESH EQU HIRESL+$1
36     SHPL EQU $50
37     SHPH EQU SHPL+$1
38     SSHPL EQU $52
39     SSHPH EQU $53
40     STESTL EQU $54
41     STESTH EQU STESTL+$1
42     BOMBL EQU $56
43     BOMBH EQU BOMBL+$1
44     PREAD EQU $FB1E
6021: AD 50 C0 45     PROG LDA $C050
6024: AD 52 C0 46     LDA $C052
6027: AD 57 C0 47     LDA $C057
602A: 20 A4 62 48     JSR CLRSCR
49     *
50     *I N I T I L I Z A T I O N
51     *
602D: A9 00 52     LDA #$00
602F: 8D 14 60 53     STA LFLAG
6032: 8D 1A 60 54     STA BMLOCK
6035: 8D 1C 60 55     STA KILL
6038: 8D 13 60 56     STA SHOT
57     *INITILIZE SCORE & PUT ON SCREEN
603B: A9 20 58     SCOREI LDA #$20
603D: 85 27 59     STA HIRESH
603F: A9 1D 60     LDA #$1D          ;LOCATION OF SCORE WORDS

```

```

6041: 85 26 61 STA HIRESL
6043: A9 05 62 LDA #$05
6045: 8D 10 60 63 STA LNGH
6048: A9 6A 64 LDA #>SCOREWD
604A: 85 51 65 STA SHPH
604C: A9 08 66 LDA #<SCOREWD
604E: 85 50 67 STA SHPL
6050: 20 E8 66 68 JSR SCOREDR ;PUT WORDS ON SCREEN
6053: A9 00 69 LDA #$00
6055: 8D 1F 60 70 STA SCOREB
6058: 8D 20 60 71 STA SCOREC
605B: A9 FF 72 LDA #$FF
605D: 8D 1E 60 73 STA SCOREA ;FIRST TIME SCORE USED WILL--
6060: 8D 1D 60 74 STA KILLNUM ;INCREMENT TO 0
6063: 20 5D 66 75 JSR SCORE
76 *INITIALIZE SHIP POSITION
6066: A9 03 77 LDA #$03
6068: 8D 12 60 78 STA SLNGH
606B: A9 D7 79 LDA #<SHIP
606D: 85 52 80 STA SSHPL
606F: A9 68 81 LDA #>SHIP
6071: 85 53 82 STA SSHPH
6073: A9 BF 83 LDA #<MSHIP
6075: 85 54 84 STA TESTL
6077: A9 68 85 LDA #>MSHIP
6079: 85 55 86 STA TESTH
607B: A9 50 87 LDA #$50
607D: 8D 0C 60 88 STA VERT
89 *INITIALIZE START OF SCROLL
6080: A9 00 90 LDA #$00
6082: 8D 04 60 91 STA INDEX
6085: 8D 05 60 92 STA PADDLEL
6088: 8D 06 60 93 STA PADDLEH
94 *
95 *M A I N P R O G R A M L O O P
96 *
97 *READ PADDLE #1
608B: A2 01 98 START LDX #$01
608D: 20 1E FB 99 JSR PREAD
6090: C0 B8 100 CPY #$B8 ;CLIP VALUE (0-183)
6092: 90 02 101 BLT SKIPP
6094: A0 B7 102 LDY #$B7
6096: 8C 07 60 103 SKIPP STY PDL
6099: 98 104 TYA
609A: CD 0C 60 105 CMP VERT ;PADDLE<VERT POS THEN SUBTRACT 5
609D: B0 1E 106 BGE PADDLE3
609F: AD 0C 60 107 LDA VERT
60A2: 38 108 SEC
60A3: E9 05 109 SBC #$05
60A5: B0 08 110 BGE PADDLE1 ;MAKE SURE =>0
60A7: A9 00 111 LDA #$00
60A9: 8D 0C 60 112 STA VERT
60AC: 8D 0D 60 113 STA TVERT
60AF: CD 07 60 114 PADDLE1 CMP PDL ;DON'T WANT TO GO PAST PADDLE POS
60B2: B0 03 115 BGE PADDLE2
60B4: AD 07 60 116 LDA PDL
60B7: 8D 0C 60 117 PADDLE2 STA VERT
60BA: 4C D3 60 118 JMP PADDLE6
60BD: CD 0C 60 119 PADDLE3 CMP VERT ;PADDLE>VERT POS THEN ADD 5
60C0: F0 0B 120 BEQ PADDLE4

```

60C2:	AD	OC	60	121		LDA	VERT	
60C5:	18			122		CLC		
60C6:	69	05		123		ADC	#\$05	
60C8:	CD	07	60	124		CMP	PDL	;DON'T WANT TO GO PAST PADDLE POS
60CB:	90	03		125		BLT	PADDLE5	
60CD:	AD	07	60	126	PADDLE4	LDA	PDL	
60D0:	8D	0C	60	127	PADDLE5	STA	VERT	
60D3:	8D	0D	60	128	PADDLE6	STA	TVERT	
60D6:	20	D3	63	129		JSR	LASER	;FIRE LASER
				130	*PUT ALIEN OBJECTS ON SCREEN AT PROPER TIMES			
60D9:	A2	00		131		LDX	#00	
60DB:	8E	0F	60	132		STX	OBJ	
60DE:	A9	69		133		LDA	#>SHAPES	;GET HI BYTE OF SHAPES
60E0:	85	51		134		STA	SHPH	
60E2:	A9	02		135	NXT	LDA	#\$02	;EACH SHAPE 2 BYTES WIDE
60E4:	8D	10	60	136		STA	LNGH	
60E7:	AE	0F	60	137		LDX	OBJ	
60EA:	BD	98	68	138		LDA	ALIVE,X	
60ED:	DO	03		139		BNE	TEST	;ALIVE?
60EF:	4C	7D	61	140		JMP	NOBJ	
60F2:	BD	A6	68	141	TEST	LDA	ONFLAG,X	
60F5:	DO	3E		142		BNE	UPDATE	;IS ONFLAG ALREADY ON?
60F7:	BD	AD	68	143		LDA	ONPOS,X	
60FA:	CD	04	60	144		CMP	INDEX	
60FD:	BO	7E		145		BGE	NOBJ	
60FF:	BD	9F	68	146		LDA	USFLAG,X	
6102:	FO	03		147		BEQ	TEST1	;IS USED ALREADY FLAG ON?
6104:	4C	7D	61	148		JMP	NOBJ	
6107:	A9	01		149	TEST1	LDA	#\$01	
6109:	9D	A6	68	150		STA	ONFLAG,X	;SET ONFLAG ON
610C:	9D	9F	68	151		STA	USFLAG,X	
610F:	A9	26		152		LDA	#\$26	
6111:	9D	8A	68	153		STA	TABLEX,X	;UPDATE TABLE
6114:	BC	B	68	154		LDY	SHPADR,X	;WHICH TYPE SHAPE
6117:	B9	BB	68	155		LDA	SHPLO,Y	;WHERE LO SHAPE IS
611A:	85	50		156		STA	SHPL	
611C:	BC	91	68	157		LDY	TABLEY,X	;GET Y POSITION
611F:	B9	0A	67	158		LDA	YVERTL,Y	
6122:	85	26		159		STA	HIRESL	
6124:	B9	CA	67	160		LDA	YVERTH,Y	
6127:	85	27		161		STA	HIRESH	
6129:	AO	26		162		LDY	#\$26	;THIS IS X=38 FAR RIGHT
612B:	98			163		TYA		
612C:	9D	8A	68	164		STA	TABLEX,X	;UPDATE TABLE
612F:	20	4E	63	165		JSR	DRAW	
6132:	4C	7D	61	166		JMP	NOBJ	
6135:	AE	0F	60	167	UPDATE	LDX	OBJ	
6138:	20	9F	63	168		JSR	DSETUP	
613B:	20	7D	63	169		JSR	XDRAW	
613E:	AE	0F	60	170		LDX	OBJ	
6141:	DE	8A	68	171		DEC	TABLEX,X	;MOVE OBJECT LEFT ONE
6144:	BD	8A	68	172		LDA	TABLEX,X	
6147:	C9	00		173		CMP	#\$00	
6149:	10	08		174		BPL	PASS	;>=0 THEN STILL ON SCREEN
614B:	A9	00		175		LDA	#\$00	
614D:	9D	A6	68	176		STA	ONFLAG,X	
6150:	4C	7D	61	177		JMP	NOBJ	
6153:	AE	0F	60	178	PASS	LDX	OBJ	
6156:	20	9F	63	179		JSR	DSETUP	
6159:	20	4E	63	180		JSR	DRAW	

615C: AD 1C 60 181		LDA KILL	
615F: C9 00 182		CMP #\$00	
6161: F0 1A 183		BEQ NOBJ	
6163: AE 0F 60 184		LDX OBJ	
6166: 20 9F 63 185		JSR DSETUP	
6169: 20 7D 63 186		JSR XDRAW	;REMOVE ALIEN
616C: AE 0F 60 187		LDX OBJ	
616F: A9 00 188		LDA #\$00	
6171: 9D 98 68 189		STA ALIVE,X	;SET OBJECT TO DEAD
6174: 9D A6 68 190		STA ONFLAG,X	;TURN OFF ON FLAG
6177: 8D 1C 60 191		STA KILL	;RESET KILL DETECTOR
617A: 20 5D 66 192		JSR SCORE	
617D: EE 0F 60 193	NOBJ	INC OBJ	;NEXT OJECT
6180: AD 0F 60 194		LDA OBJ	
6183: C9 07 195		CMP #\$07	
6185: F0 03 196		BEQ TEST2	;DONE WITH ALL?
6187: 4C E2 60 197		JMP NXT	
618A: EE 04 60 198	TEST2	INC INDEX	;UPDATE SCROLL COUNTER
618D: AD 04 60 199		LDA INDEX	
6190: D0 0C 200		BNE PASS1	
6192: A0 00 201		LDY #00	;RESET ALL ALREADY USED FLAGS TO 0
6194: A9 00 202	AGAIN	LDA #\$00	
6196: 99 9F 68 203		STA USFLAG,Y	
6199: C8 204		INY	
619A: C0 08 205		CPY #\$08	
619C: D0 F6 206		BNE AGAIN	
619E: 20 33 63 207	PASS1	JSR SSETUP	
61A1: 20 BE 62 208		JSR SDRAW	
61A4: 20 89 64 209		JSR BOMB	
61A7: 20 01 62 210		JSR SCROLL	
211		*DETECT FOR MT COLLISION	
61AA: AD 05 60 212		LDA PADDLEL	
61AD: 18 213		CLC	
61AE: 69 55 214		ADC #\$55	;TIP OF SHIP @84
61B0: 85 09 215		STA \$09	
61B2: AD 06 60 216		LDA PADDLEH	
61B5: 69 40 217		ADC #\$40	;LOCATION OF MOUNTAIN TABLE
61B7: 85 0A 218		STA \$0A	
61B9: A0 00 219		LDY #\$00	
61BB: 18 220		CLC	
61BC: AD 0C 60 221		LDA VERT	
61BF: 69 08 222		ADC #\$08	;BECAUSE PDL IS AT TOP OF PLANE---
61C1: 8D 08 60 223		STA TEMP	;AND MOUNTAINS HIT BOTTOM
61C4: B1 09 224		LDA (\$09),Y	
61C6: CD 08 60 225		CMP TEMP	
61C9: B0 03 226		BGE NOHIT	
61CB: 4C 13 65 227		JMP EXPLODE	
61CE: AD 0C 60 228	NOHIT	LDA VERT	
61D1: 8D 0D 60 229		STA TVERT	
61D4: 20 33 63 230		JSR SSETUP	
61D7: 20 FD 62 231		JSR SXDRAW	
61DA: 20 14 64 232	FIN	JSR XLASER	
61DD: 20 F7 64 233		JSR BOMBX	
234		*TEST IF ALL ALIENS KILLED AND RESET WHEN INDEX=0	
61E0: AD 1D 60 235	RSETAL	LDA KILLNUM	
61E3: C9 07 236		CMP #\$07	
61E5: D0 16 237		BNE RSETAL2	
61E7: AD 04 60 238		LDA INDEX	;CHECK IF START OF TERRAIN
61EA: D0 11 239		BNE RSETAL2	
61EC: A9 00 240		LDA #\$00	;RESET

```

61EE: 8D 1D 60 241      STA KILLNUM
61F1: A2 00      242      LDX #$00
61F3: A9 01      243      LDA #$01
61F5: 9D 98 68 244      RSETAL1 STA ALIVE,X
61F8: E8          245      INX
61F9: E0 07      246      CPX #$07
61FB: D0 F8      247      BNE RSETAL1
61FD: EA          248      RSETAL2 NOP
61FE: 4C 8B 60 249      JMP START
250 *
251 *S U B R O U T I N E S *****
252 *
253 *SCROLLING ROUTINE SETUP
254 *
6201: AD 04 60 255      SCROLL  LDA INDEX      ;COUNTER FOR WHERE YOU ARE INTO
256 *-                ;TERRAIN
6204: C9 00      257      CMP #$00      ;IF ZERO RESET GROUND TABLE POINTER
6206: F0 11      258      BEQ RSET
6208: 18          259      CLC
6209: AD 05 60 260      LDA PADDLEL      ;EACH CYCLE ADVANCE 7 MORE INTO --
620C: 69 07      261      ADC #$07      ;GROUND ARRAY
620E: 8D 05 60 262      STA PADDLEL
6211: 90 03      263      BCC C
6213: EE 06 60 264      INC PADDLEH
6216: 4C 21 62 265      C          JMP SCONT
6219: A9 00      266      RSET  LDA #$00      ;RESET GROUND POSITION BACK TO 0
621B: 8D 05 60 267      STA PADDLEL
621E: 8D 06 60 268      STA PADDLEH
269 *
270 *SCROLLING ROUTINE
271 *
6221: A9 02      272      SCONT  LDA #$02
6223: 8D 03 60 273      STA COUNT      ;COUNTER SO DRAWS 1ST TIME
6226: A9 01      274      ERASE  LDA #$01
6228: 85 08      275      STA $08      ;BIT COUNTER
622A: A9 00      276      LDA #$00      ; START OF ARRAY LO BYTE
622C: 85 06      277      STA $06
622E: A9 40      278      LDA #$40      ; START OF ARRAY HI BYTE
6230: 85 07      279      STA $07
6232: AD 05 60 280      LDA PADDLEL      ;OFFSET INTO ARRAY LO BYT
6235: 85 04      281      STA $04
6237: AD 06 60 282      LDA PADDLEH      ;OFFSET HI BYTE
623A: 29 07      283      AND #$07      ;SO NOT BEYOND TABLE
623C: 85 05      284      STA $05
623E: A2 00      285      LDX #$00
6240: 18          286      LOOP   CLC
6241: A5 04      287      LDA $04      ;OFFSET INTO TABLE (LO)
6243: 65 06      288      ADC $06      ;ADD BASE ADDRESS (LO)
6245: 85 02      289      STA $02
6247: A5 05      290      LDA $05      ; (HI)
6249: 65 07      291      ADC $07
624B: 85 03      292      STA $03      ;REG 2&3 ACTUAL ADDRESS OF SPECI-
293 *-                ;FIC BYTE IN TABLE
624D: A0 00      294      LDY #$00
624F: B1 02      295      LDA ($02),Y      ;ACTUAL VALUE AT THAT BYTE
6251: A8          296      TAY
6252: B9 0A 67 297      LDA YVERTL,Y      ;ADDRESS OF LINE ON SCREEN (LO)
6255: 85 02      298      STA $02
6257: B9 CA 67 299      LDA YVERTH,Y      ; (HI)
625A: 85 03      300      STA $03

```

625C: 8A	301	TXA		;X IS OFFSET INTO HI-RES LINE
625D: A8	302	TAY		
625E: B1 02	303	LDA (\$02),Y		;CONTAINS ADDRESS OF BEGINNING LINE
	304	*-		;NOW OFFSET INTO LINE
6260: 45 08	305	EOR \$08		;NOW LEFT HAND DOT ON
6262: 91 02	306	STA (\$02),Y		
6264: E6 04	307	INC \$04		;INCREMENT OFFSET FOR NEXT DOT (LO)
6266: D0 09	308	BNE SKIP		;IF HAVEN'T CROSSED 256 THEN SKIP
6268: 18	309	CLC		
6269: A5 05	310	LDA \$05		;INC. HI ORDER OFFSET FOR NEXT DOT
626B: 69 01	311	ADC #\$01		
626D: 29 07	312	AND #\$C7		;MAKES WRAP AROUND INTO TABLE--
626F: 85 05	313	STA \$05		;(IF HIT END OF TABLE)
6271: 06 08	314	ASL \$08		;SHIFT LEFT INTO BYTE FOR NEXT
	315	*-		;DOT TO PLOT
6273: 10 CB	316	BPL LOOP		;IF INTO BIT 7 THEN TOO FAR SO
	317	*-		;RESTORE TO 1
6275: A9 01	318	LDA #\$01		;RESTORE BIT COUNTER TO 1
6277: 85 08	319	STA \$08		
6279: E8	320	INX		;NEXT BYTE BECAUSE HAVE ALREADY
	321	*-		;DONE 7 DOTS
627A: E0 28	322	CPX #\$28		;SEE IF COMPLETELY ACROSS 40 BYTES
627C: D0 C2	323	BNE LOOP		
627E: CE 03 60	324	DEC COUNT		
6281: AD 03 60	325	LDA COUNT		
6284: C9 01	326	CMP #\$01		;IF=1 ONLY HAVE DRAWN TERRAIN
6286: 90 1B	327	BLT SKIP1		;TERRAIN ALREADY DRAWN&XDRAWN,DONE
	328	*		
	329	*SINGLE STEP DEBUG PACKAGE		
	330	*		
6288: AD 00 CO	331	LDA \$C000		;KEY PRESSED?
628B: 10 10	332	BPL IGNORE		;EXIT IF NO KEY PRESSED
628D: C9 9B	333	CMP #\$9B		;ESC KEY?
628F: D0 OC	334	BNE IGNORE		
6291: 2C 10 CO	335	CAUGHT BIT \$C010		;CLEAR STROBE
6294: AD 00 CO	336	LDA \$C000		;KEY PRESSED
6297: 10 FB	337	BPL *-3		;LOOP BY BRANCHING BACK 3 BYTES
6299: C9 AO	338	CMP #\$AO		;SPACE KEY?
629B: D0 03	339	BNE IGNORE+3		;NO DON'T CLEAR STROBE
629D: 2C 10 CO	340	IGNORE BIT \$C010		;CLEAR STROBE
	341	*		
62A0: 4C 26 62	342	JMP ERASE		;ONLY DRAWN SO FAR; NOW GO TO ERAS
	343	*-		;TO DRAW AGAIN
62A3: 60	344	SKIP1 RTS		
	345	*		
	346	*CLEAR SCREEN SUBROUTINE		
	347	*		
62A4: A9 00	348	CLRSCR LDA #\$00		
62A6: 85 26	349	STA HIRESL		
62A8: A9 20	350	LDA #\$20		
62AA: 85 27	351	STA HIRESH		
62AC: AO 00	352	CLR1 LDY #\$00		
62AE: A9 00	353	LDA #\$00		
62B0: 91 26	354	CLR2 STA (HIRESL),Y		
62B2: C8	355	INY		
62B3: D0 FB	356	BNE CLR2		
62B5: E6 27	357	INC HIRESH		
62B7: A5 27	358	LDA HIRESH		
62B9: C9 40	359	CMP #\$40		
62BB: 90 EF	360	BCC CLR1		

```

62BD: 60      361      RTS
362      *
363      *DRAW SHIP SUBROUTINE
364      *DRAW SHAPE ONE LINE AT A TIME-LNGH BYTES ACROSS
365      *
62BE: A9 00    366      SDRAW      LDA      #$00
62CO: 8D 15 60 367      STA      ESET
62C3: AC 0D 60 368      SDRAW1     LDY      TVERT      ;VERTICAL POSITION
62C6: 20 1C 63 369      JSR      GETADR
62C9: A2 00    370      LDX      #$00
62CB: A1 54    371      SDRAW2     LDA      (STESTL,X) ;GET BYTE OF SHIP MASK SHAPE
62CD: 29 7F    372      AND      #$7F      ;MASK OUT HI BIT
62CF: 31 26    373      AND      (HIRESL),Y ;(AND) IT AGAINST SCREEN
62D1: C9 00    374      CMP      #$00      ; IF ANYTHING IN WAY GET>0
62D3: F0 05    375      BEQ      SDRAW3
62D5: A9 01    376      LDA      #$01      ;SET BECAUSE IF DON'T FINISH DRAW-
62D7: 8D 15 60 377      STA      ESET      ;ING SHIP,PIECE LEFT WHEN XDRAW
378      *                      ;DURING EXPLOSION
62DA: A1 52    379      SDRAW3     LDA      (SSHPL,X) ;GET BYTE OF SHIP'S SHAPE
62DC: 51 26    380      EOR      (HIRESL),Y
62DE: 91 26    381      STA      (HIRESL),Y ;PLOT
62E0: E6 54    382      INC      STESTL      ;NEXT BYTE OF MASK
62E2: E6 52    383      INC      SSHPL      ; NEXT BYTE OF TABLE
62E4: C8       384      INY              ;NEXT SCREEN POSITION
62E5: CE 12 60 385      DEC      SLNGH
62E8: DO E1    386      BNE      SDRAW2      ;IF LINE NOT FINISHED BRANCH
62EA: EE 0D 60 387      INC      TVERT      ;OTHERWISE NEXT LINE DOWN
62ED: CE 11 60 388      DEC      DEPTH
62F0: DO D1    389      BNE      SDRAW1      ;DONE DRAWING?
62F2: AD 15 60 390      LDA      ESET      ;IS EXPLOSION FLAG SET?
62F5: C9 00    391      CMP      #$00
62F7: FO 03    392      BEQ      SDRAW4      ;NO!, EXIT
62F9: 4C 13 65 393      JMP      EXLODE      ;YES!, EXPLODE SHIP
62FC: 60      394      SDRAW4     RTS
395      *
396      *XDRAW SHIP SUBROUTINE
397      *
62FD: AC 0D 60 398      SXDRAW     LDY      TVERT      ;PADDLE VALUE
6300: 20 1C 63 399      JSR      GETADR
6303: A2 00    400      LDX      #$00
6305: A1 52    401      SXDRAW2     LDA      (SSHPL,X)
6307: 51 26    402      EOR      (HIRESL),Y
6309: 91 26    403      STA      (HIRESL),Y
630B: E6 52    404      INC      SSHPL
630D: C8       405      INY
630E: CE 12 60 406      DEC      SLNGH
6311: DO F2    407      BNE      SXDRAW2
6313: EE 0D 60 408      INC      TVERT
6316: CE 11 60 409      DEC      DEPTH
6319: DO E2    410      BNE      SXDRAW
631B: 60      411      RTS
412      *
413      *GETADR SUBROUTINE
414      *
631C: B9 0A 67 415      GETADR      LDA      YVERTL,Y ;LOOK UP LO BYTE OF LINE
631F: 18       416      CLC
6320: 6D 0E 60 417      ADC      HORIZ      ;ADD DISPLACEMENT INTO LINE
6323: 85 26    418      STA      HIRESL
6325: B9 CA 67 419      LDA      YVERTH,Y ;LOOK UP HI BYTE OF LINE
6328: 85 27    420      STA      HIRESH

```



```

632A: AD 08 60 421      LDA  TEMP
632D: 8D 12 60 422      STA  SLNGH      ;RESTORE VARIABLE
6330: A0 00      423      LDY  #$00
6332: 60      424      RTS
      425      *
      426      *SHIP SET UP SUBROUTINE
      427      *
6333: A9 D7      428      SSETUP  LDA  #<SHIP      ;SHAPE TABLE LOCATION
6335: 85 52      429      STA  SSHPL
6337: A9 68      430      LDA  #>SHIP
6339: 85 53      431      STA  SSHPH
633B: A9 08      432      LDA  #$08
633D: 8D 11 60 433      STA  DEPTH
6340: A9 09      434      LDA  #$09
6342: 8D 0E 60 435      STA  HORIZ
6345: A9 03      436      LDA  #$03
6347: 8D 12 60 437      STA  SLNGH
634A: 8D 08 60 438      STA  TEMP
634D: 60      439      RTS
      440      *
      441      *DRAW ALIEN SHIPS & TARGETS SUBROUTINE
      442      *DRAW SHAPE ONE COLUMN AT A TIME
      443      *
634E: A2 00      444      DRAW    LDX  #$00
6350: A1 50      445      DRAW2   LDA  (SHPL,X)
6352: 29 7F      446      AND    #$7F      ;MASK OUT HI BIT
6354: 31 26      447      AND    (HIRESL),Y ;(AND) IT AGAINST SCREEN
6356: C9 00      448      CMP    #$00      ;IF ANYTHING IN WAY GET>0
6358: F0 03      449      BEQ    DRAW3      ;NO COLLISION, BRANCH TO DRAW3
635A: EE 1C 60 450      INC    KILL      ;COLLISION! INCREMENT KILL
635D: A1 50      451      DRAW3   LDA  (SHPL,X)
635F: 51 26      452      EOR    (HIRESL),Y ;(EOR) WITH SCREEN
6361: 91 26      453      STA    (HIRESL),Y ;PLOT
6363: A5 27      454      LDA    HIRESH
6365: 18      455      CLC
6366: 69 04      456      ADC    #$04
6368: 85 27      457      STA    HIRESH
636A: E6 50      458      INC    SHPL
636C: C9 40      459      CMP    #$40
636E: 90 F0      460      BCC    DRAW2
6370: E9 20      461      SBC    #$20
6372: 85 27      462      STA    HIRESH
6374: CE 10 60 463      DEC    LNGH
6377: F0 03      464      BEQ    DRAW4
6379: C8      465      INY
637A: D0 D4      466      BNE    DRAW2
637C: 60      467      DRAW4   RTS
      468      *
      469      *XDRAW ALIEN SHIPS & TARGETS SUBROUTINE
      470      *
637D: A2 00      471      XDRAW   LDX  #$00
637F: A1 50      472      XDRAW2  LDA  (SHPL,X)
6381: 51 26      473      EOR    (HIRESL),Y
6383: 91 26      474      STA    (HIRESL),Y
6385: A5 27      475      LDA    HIRESH
6387: 18      476      CLC
6388: 69 04      477      ADC    #$04
638A: 85 27      478      STA    HIRESH
638C: E6 50      479      INC    SHPL
638E: C9 40      480      CMP    #$40

```

```

6390: 90 ED 481 BCC XDRAW2
6392: E9 20 482 SBC #$20
6394: 85 27 483 STA HIRESH
6396: CE 10 60 484 DEC LNGH
6399: F0 03 485 BEQ XDRAW3
639B: C8 486 INY
DRAW2
639E: 60 488 XDRAW3 RTS
489 *
490 *DRAWING ROUTINES SETUP
491 *
639F: BC 91 68 492 DSETUP LDY TABLEY,X
63A2: B9 0A 67 493 LDA YVERTL,Y
63A5: 85 26 494 STA HIRESL
63A7: B9 CA 67 495 LDA YVERTH,Y
63AA: 85 27 496 STA HIRESH
63AC: A9 02 497 LDA #$02
63AE: 8D 10 60 498 STA LNGH
63B1: 18 499 CLC
63B2: BD 8A 68 500 LDA TABLEX,X
63B5: 4A 501 LSR
63B6: B0 0B 502 BCS ODD ;TEST FOR EVEN OR ODD OFFSET FROM
503 *- ;X VALUE IN TABLEX
63B8: BC B4 68 504 EVEN LDY SHPADR,X
63BB: B9 BB 68 505 LDA SHPLO,Y
63BE: 85 50 506 STA SHPL
63C0: 4C CB 63 507 JMP GOON
63C3: BC B4 68 508 ODD LDY SHPADR,X
63C6: B9 BD 68 509 LDA SHPLO+2,Y
63C9: 85 50 510 STA SHPL
63CB: BC 8A 68 511 GOON LDY TABLEX,X
63CE: A9 69 512 LDA #>SHAPES
63D0: 85 51 513 STA SHPH
63D2: 60 514 RTS
515 *
516 *LASER SUBROUTINE
517 *
63D3: AD 62 C0 518 LASER LDA $C062 ;NEG IF BUTTON PRESSED
63D6: 30 08 519 BMI FIRE1
63D8: A9 00 520 LDA #$00 ;BUTTON NOT PRESSED,SET FLAG TO 0
63DA: 8D 14 60 521 STA LFLAG
63DD: 4C 13 64 522 JMP NOSHOT
63E0: AD 14 60 523 FIRE1 LDA LFLAG ;IS BUTTON BEING HELD DOWN?
63E3: C9 01 524 CMP #$01
63E5: B0 2C 525 BGE NOSHOT
63E7: A9 01 526 LDA #$01
63E9: 8D 13 60 527 STA SHOT ;SET LASER FIRED FLAG
63EC: 8D 14 60 528 STA LFLAG ;SET BUTTON PRESSED FLAG
63EF: 18 529 CLC
63F0: AD 0C 60 530 LDA VERT ;TOP OF SHIP
63F3: 69 07 531 ADC #$07
63F5: A8 532 TAY ;Y REG CONTAINS VERT. LSER POS.
63F6: A9 0C 533 LDA #$0C ;START AT HORIZ=$0C
63F8: 8D 0E 60 534 STA HORIZ
63FB: 20 1C 63 535 JSR GETADR ;FIND ADDRESS OF LASER BEAM LINE
63FE: A2 0E 536 LDX #$0E ;SET UP LOOP FOR E TIMES
6400: A9 AA 537 LASER1 LDA #$AA ;DRAW PAIRS OF AA & D5 BYTES(RED)
6402: 51 26 538 EOR (HIRESL),Y ;BY ORING AGAINST SCREEN
6404: 91 26 539 STA (HIRESL),Y
6406: E6 26 540 INC HIRESL ;NEXT SCREEN POSITION

```

6408:	A9 D5	541	LDA	#\$D5	
640A:	51 26	542	EOR	(HIRESL),Y	
640C:	91 26	543	STA	(HIRESL),Y	
640E:	E6 26	544	INC	HIRESL	;NEXT SCREEN POSITION
6410:	CA	545	DEX		;DECREMENT INDEX TO LOOP
6411:	DO ED	546	BNE	LASER1	;DONE?
6413:	60	547	NOSHOT	RTS	;YES! EXIT
		548	*XDRAW	LASER	SUBROUTINE
6414:	AD 13 60	549	XLASER	LDA	SHOT
6417:	C9 01	550	CMP	#\$01	;HAS LASER BEEN SHOT?
6419:	DO 24	551	BNE	NXSHOT	;NO! SKIP XDRAWING LASER
641B:	18	552	CLC		
641C:	AD 0C 60	553	LDA	VERT	
641F:	69 07	554	ADC	#\$07	
6421:	A8	555	TAY		
6422:	A9 0C	556	LDA	#\$0C	
6424:	8D 0E 60	557	STA	HORIZ	
6427:	20 1C 63	558	JSR	GETADR	
642A:	A2 0E	559	LDX	#\$0E	
642C:	A9 AA	560	LASER2	LDA	#\$AA
642E:	51 26	561	EOR	(HIRESL),Y	
6430:	91 26	562	STA	(HIRESL),Y	
6432:	E6 26	563	INC	HIRESL	
6434:	A9 D5	564	LDA	#\$D5	
6436:	51 26	565	EOR	(HIRESL),Y	
6438:	91 26	566	STA	(HIRESL),Y	
643A:	E6 26	567	INC	HIRESL	
643C:	CA	568	DEX		
643D:	DO ED	569	BNE	LASER2	
643F:	A9 00	570	NXSHOT	LDA	#\$00 ;RESET LASER FIRED FLAG TO OFF
6441:	8D 13 60	571	STA	SHOT	
6444:	60	572	RTS		
		573	*		
		574	*DRAWING	ROUTINES FOR BOMB	
		575	*		
6445:	A9 EF	576	BSET	LDA	#<SHBOMB ;ADDRESS BOMB SHAPE
6447:	85 56	577	STA	BOMBL	
6449:	A9 68	578	LDA	#>SHBOMB	
644B:	85 57	579	STA	BOMBH	
644D:	AD 19 60	580	LDA	BHORIZ	;BOMB'S HORIZ. POSITION
6450:	8D 0E 60	581	STA	HORIZ	
6453:	A9 03	582	LDA	#\$03	
6455:	8D 11 60	583	STA	DEPTH	
6458:	60	584	RTS		
6459:	AC 17 60	585	BDRAW	LDY	TBVERT ;BOMB VERT POS
645C:	20 1C 63	586	JSR	GETADR	
645F:	A2 00	587	LDX	#\$00	
6461:	A1 56	588	LDA	(BOMBL,X)	;GET ADDRESS OF BOMB SHAPE
6463:	91 26	589	STA	(HIRESL),Y	;PLOT
6465:	EE 17 60	590	INC	TBVERT	
6468:	E6 56	591	INC	BOMBL	
646A:	CE 11 60	592	DEC	DEPTH	
646D:	DO EA	593	BNE	BDRAW	
646F:	60	594	RTS		
6470:	AC 17 60	595	BXDRAW	LDY	TBVERT
6473:	20 1C 63	596	JSR	GETADR	
6476:	A2 00	597	LDX	#\$00	
6478:	A1 56	598	LDA	(BOMBL,X)	
647A:	51 26	599	EOR	(HIRESL),Y	
647C:	91 26	600	STA	(HIRESL),Y	

```

647E: EE 17 60 601      INC TBVERT
6481: E6 56 602      INC BOMBL
6483: CE 11 60 603      DEC DEPTH
6486: DO E8 604      BNE BXDRAW
6488: 60 605      RTS
606      *
607      *BOMB SUBROUTINE
608      *
6489: AD 61 C0 609      BOMB      LDA $C061      ;NEG IF BUTTON PRESSED
648C: 30 03 610      BMI BOMB1
648E: 4C BD 64 611      JMP NODROP
6491: AD 1A 60 612      BOMB1     LDA BMLOCK
6494: C9 01 613      CMP #$01      ;IS BOMB STILL FALLING?
6496: B0 2A 614      BGE FALLIN  ;YES, GOTO FALLIN
6498: AD 0C 60 615      DROP      LDA VERT
649B: 18 616      CLC
649C: 69 09 617      ADC #$09
649E: 8D 16 60 618      STA BVERT      ;INITIAL POSITION OF BOMB
64A1: 8D 17 60 619      STA TBVERT
64A4: A9 0A 620      LDA #$0A      ;STARTING HORIZ POSITION
64A6: 8D 19 60 621      STA BHORIZ
64A9: A9 00 622      LDA #$00      ;INITIAL VERTICAL VELOCITY
64AB: 8D 18 60 623      STA BVELY
64AE: A9 01 624      LDA #$01
64B0: 8D 1A 60 625      STA BMLOCK      ;RESET TO ON
64B3: 8D 1B 60 626      STA TBMLOCK      ;RESET END OF FALL TO OFF
64B6: 20 45 64 627      JSR BSET
64B9: 20 59 64 628      JSR BDRAW      ;DRAW BOMB
64BC: 60 629      RTS
64BD: AD 1A 60 630      NODROP     LDA BMLOCK
64C0: F0 34 631      BEQ BOMB3      ;IS BOMB STILL FALLING
64C2: AD 18 60 632      FALLIN     LDA BVELY
64C5: 18 633      CLC
64C6: 69 05 634      ADC #$05      ;ADD ACCELERATION CONSTANT
64C8: 8D 18 60 635      STA BVELY      ;NEW VERTICAL VELOCITY
64CB: 6D 16 60 636      ADC BVERT
64CE: 8D 17 60 637      STA TBVERT
64D1: 8D 16 60 638      STA BVERT      ;BOMB'S NEW VERTICAL POSITION
64D4: AD 19 60 639      LDA BHORIZ
64D7: 69 01 640      ADC #$01      ;BOMB'S HORIZ. VELOCITY(CONSTANT)
64D9: 8D 19 60 641      STA BHORIZ      ;BOMB'S NEW HORIZ. POSITION
642      *TEMP DETECT FOR BOMB LANDING
64DC: AD 16 60 643      LDA BVERT
64DF: C9 B0 644      CMP #$B0      ;BOTTOM SCREEN?
64E1: 90 0D 645      BLT BOMB2      ;NO! THEN BOMB2
64E3: A9 B0 646      LDA #$B0
64E5: 8D 16 60 647      STA BVERT
64E8: 8D 17 60 648      STA TBVERT
64EB: A9 00 649      LDA #$00
64ED: 8D 1B 60 650      STA TBMLOCK      ;SET END OF BOMB FALL FLAG
64F0: 20 45 64 651      BOMB2     JSR BSET
64F3: 20 59 64 652      JSR BDRAW
64F6: 60 653      BOMB3     RTS
654      *BOMB XDRAW
64F7: AD 1A 60 655      BOMBX     LDA BMLOCK      ;IS BOMB STILL FALLING?(1=YES)
64FA: F0 16 656      BEQ BOMBX1      ;SKIP IF 0
64FC: 20 45 64 657      JSR BSET
64FF: AD 16 60 658      LDA BVERT
6502: 8D 17 60 659      STA TBVERT
6505: 20 70 64 660      JSR BXDRAW      ;XDRAW BOMB

```

```

6508: AD 1B 60 661      LDA  TBMLOCK
650B: DO 05 662      BNE  BOMBX1
650D: A9 00 663      LDA  #$00
650F: 8D 1A 60 664      STA  BMLOCK      ;RESET BOMB FALLING TO OFF
6512: 60 665      BOMBX1 RTS
        666      *
        667      *EXPLOSION SUBROUTINE
        668      *
6513: 20 1E 65 669      EXPLODE JSR  EXPSUB
6516: A9 FE 670      LDA  #$FE
6518: 20 A8 FC 671      JSR  $FCA8
651B: 4C DA 61 672      JMP  FIN
651E: AD 0C 60 673      EXPSUB  LDA  VERT
6521: 8D OD 60 674      STA  TVERT
6524: 20 33 63 675      JSR  SSETUP      ;XDRAW SHIP
6527: 20 FD 62 676      JSR  SXDRAW
652A: A9 04 677      EDRAW   LDA  #$04      ;PLOT WHITE FIREBALL 4 LINES DEEP
652C: 8D 11 60 678      STA  DEPTH
652F: A9 0A 679      LDA  #$0A      ;HORIZ POS SHIP'S CENTER
6531: 8D OE 60 680      STA  HORIZ
6534: AD 0C 60 681      LDA  VERT      ;VERT POS TOP OF SHIP
6537: 18 682      CLC
6538: 69 04 683      ADC  #$04      ;TO REACH CENTER
653A: 8D OD 60 684      STA  TVERT
653D: AC OD 60 685      EDRAW1  LDY  TVERT      ;SHIP'S CENTER
6540: 20 1C 63 686      JSR  GETADR
6543: A9 FF 687      LDA  #$FF      ;WHITE LINE
6545: 51 26 688      EOR  (HIRESL),Y
6547: 91 26 689      STA  (HIRESL),Y
6549: EE OD 60 690      INC  TVERT      ;NEXT LINE
654C: CE 11 60 691      DEC  DEPTH
654F: DO EC 692      BNE  EDRAW1      ;DONE?
6551: A9 80 693      LDA  #$80
6553: 20 A8 FC 694      JSR  $FCA8      ;DELAY
        695      *XDRAW SEQ1 -8 BLOCKS
6556: A9 00 696      LDA  #$00
6558: 8D OA 60 697      STA  SBLOCK
655B: A9 08 698      LDA  #$08
655D: 8D OB 60 699      STA  EBLOCK
6560: 20 1A 66 700      JSR  EPLOT
        701      *XDRAW BEGINING FLASH
6563: A9 04 702      EDRAW2  LDA  #$04
6565: 8D 11 60 703      STA  DEPTH
6568: A9 0A 704      LDA  #$0A
656A: 8D OE 60 705      STA  HORIZ
656D: 18 706      CLC
656E: AD 0C 60 707      LDA  VERT
6571: 69 04 708      ADC  #$04
6573: 8D OD 60 709      STA  TVERT
6576: AC OD 60 710      EDRAW3  LDY  TVERT
6579: 20 1C 63 711      JSR  GETADR
657C: B1 26 712      LDA  (HIRESL),Y
657E: 51 26 713      EOR  (HIRESL),Y
6580: 91 26 714      STA  (HIRESL),Y
6582: EE OD 60 715      INC  TVERT
6585: CE 11 60 716      DEC  DEPTH
6588: DO EC 717      BNE  EDRAW3
        718      *XDRAW SEQ2-11BLOCKS
658A: A9 08 719      LDA  #$08
658C: 8D OA 60 720      STA  SBLOCK

```

658F:	A9 13	721	LDA	#\$13
6591:	8D 0B 60	722	STA	EBLOCK
6594:	20 1A 66	723	JSR	EPL0T
		724	*XDRAW	SEQ1- 8 OFF
6597:	A9 00	725	LDA	#\$00
6599:	8D 0A 60	726	STA	SBLOCK
659C:	A9 08	727	LDA	#\$08
659E:	8D 0B 60	728	STA	EBLOCK
65A1:	20 1A 66	729	JSR	EPL0T
		730	*XDRAW	SEQ3-15
65A4:	A9 13	731	LDA	#\$13
65A6:	8D 0A 60	732	STA	SBLOCK
65A9:	A9 22	733	LDA	#\$22
65AB:	8D 0B 60	734	STA	EBLOCK
65AE:	20 1A 66	735	JSR	EPL0T
		736	*XDRAW	SEQ2-11 OFF
65B1:	A9 08	737	LDA	#\$08
65B3:	8D 0A 60	738	STA	SBLOCK
65B6:	A9 13	739	LDA	#\$13
65B8:	8D 0B 60	740	STA	EBLOCK
65BB:	20 1A 66	741	JSR	EPL0T
		742	*XDRAW	SEQ4-16
65BE:	A9 22	743	LDA	#\$22
65C0:	8D 0A 60	744	STA	SBLOCK
65C3:	A9 32	745	LDA	#\$32
65C5:	8D 0B 60	746	STA	EBLOCK
65C8:	20 1A 66	747	JSR	EPL0T
		748	*XDRAW	SEQ3-15 OFF
65CB:	A9 13	749	LDA	#\$13
65CD:	8D 0A 60	750	STA	SBLOCK
65D0:	A9 22	751	LDA	#\$22
65D2:	8D 0B 60	752	STA	EBLOCK
65D5:	20 1A 66	753	JSR	EPL0T
		754	*XDRAW	SEQ5- 18
65D8:	A9 32	755	LDA	#\$32
65DA:	8D 0A 60	756	STA	SBLOCK
65DD:	A9 44	757	LDA	#\$44
65DF:	8D 0B 60	758	STA	EBLOCK
65E2:	20 1A 66	759	JSR	EPL0T
		760	*XDRAW	SEQ4-16 OFF
65E5:	A9 22	761	LDA	#\$22
65E7:	8D 0A 60	762	STA	SBLOCK
65EA:	A9 32	763	LDA	#\$32
65EC:	8D 0B 60	764	STA	EBLOCK
65EF:	20 1A 66	765	JSR	EPL0T
		766	*XDRAW	SEQ6-18
65F2:	A9 44	767	LDA	#\$44
65F4:	8D 0A 60	768	STA	SBLOCK
65F7:	A9 56	769	LDA	#\$56
65F9:	8D 0B 60	770	STA	EBLOCK
65FC:	20 1A 66	771	JSR	EPL0T
		772	*XDRAW	SEQ5-18 OFF
65FF:	A9 32	773	LDA	#\$32
6601:	8D 0A 60	774	STA	SBLOCK
6604:	A9 44	775	LDA	#\$44
6606:	8D 0B 60	776	STA	EBLOCK
6609:	20 1A 66	777	JSR	EPL0T
		778	*XDRAW	SEQ6-18 OFF
660C:	A9 44	779	LDA	#\$44
660E:	8D 0A 60	780	STA	SBLOCK

```

6611: A9 56 781 LDA #$56
6613: 8D 0B 60 782 STA EBLOCK
6616: 20 1A 66 783 JSR EPL0T
6619: 60 784 RTS
785 *
786 *EXPLOSION PLOTTING SUBROUTINE
787 *
661A: AE 0A 60 788 EPL0T LDX SBLOCK ;LOCATION IN PARTICLE POSITION
789 *- ;TO START DRAWING
661D: A9 03 790 EPL0T1 LDA #$03 ;EACH BLOCK 3 LINES DEEP
661F: 8D 11 60 791 STA DEPTH
6622: 18 792 ELOOP1 CLC
6623: AD 0C 60 793 LDA VERT ;TOP OF SHIP
6626: 69 04 794 ADC #$04 ;NOW CENTER OF SHIP
6628: 18 795 CLC
6629: 7D 9A 69 796 ADC EOFFY,X ;ADD RELATIVE Y POS OF PARTICLE.
662C: C9 00 797 CMP #$00 ;TEST NOT OFF TOP SCREEN
662E: 90 21 798 BLT NOPLOT ;IF OFF, DON'T LOT
6630: C9 C0 799 CMP #$C0 ;TEST NOT OFF BOTTOM SCREEN
6632: B0 1D 800 BGE NOPLOT ;IF OFF, DON'T PLOT
6634: 8D 09 60 801 STA TEMP1 ;STORE VALUE IN TEMP1
6637: BD 44 69 802 LDA EOFFX,X ;LOCATE X POSITION
663A: 8D 0E 60 803 STA HORIZ
663D: AC 09 60 804 ELOOP3 LDY TEMP1 ;FIND LINE ADRESS TO PLOT ON SCREEN
6640: 20 1C 63 805 JSR GETADR
6643: A9 F0 806 LDA #$F0 ;VALUE OF ALL SHAPE BYTES
6645: 51 26 807 EOR (HIRESL),Y ;XOR WITH SCREEN
6647: 91 26 808 STA (HIRESL),Y ;PLOT ON SCREEN
6649: CE 09 60 809 DEC TEMP1 ;NEXT LINE, IN THIS CASE DRAWING --
664C: CE 11 60 810 DEC DEPTH ;FROM BOTTOM TO TOP
664F: D0 EC 811 BNE ELOOP3 ;DONE?
6651: E8 812 NOPLOT INX ;DO NEXT PARTICLE
6652: EC 0B 60 813 CPX EBLOCK ;DONE WITH ALL PARTICLES IN GROUP?
6655: D0 C6 814 BNE EPL0T1 ;NO,CONTINUE
6657: A9 30 815 LDA #$30
6659: 20 A8 FC 816 JSR $FCA8 ;DELAY
665C: 60 817 RTS
818 *
819 *SCORE SUBROUTINE
820 *
665D: EE 1D 60 821 SCORE INC KILLNUM ;ANOTHER ALIEN KILLED
6660: EE 1E 60 822 INC SCOREA ;INCREMENT COUNTER
6663: AD 1E 60 823 LDA SCOREA
6666: C9 0A 824 CMP #$0A
6668: 90 29 825 BLT SCRSET ;IF <10 DON'T CARRY TENS DIGIT
666A: A9 00 826 LDA #$00 ;ZERO OUT 1'S DIGIT
666C: 8D 1E 60 827 STA SCOREA
666F: EE 1F 60 828 SCORE10 INC SCOREB ;ADD CARRY IN TENS
6672: AD 1F 60 829 LDA SCOREB
6675: C9 0A 830 CMP #$0A
6677: 90 1A 831 BLT SCRSET ;IF <10 DON'T CARRY TO 100'S DIGIT
6679: A9 00 832 LDA #$00 ;ZERO OUT 10'S DIGIT & 1'S DIGIT
667B: 8D 1F 60 833 STA SCOREB
667E: EE 20 60 834 INC SCORC ;ADD CARRY IN 100'S
6681: AD 20 60 835 LDA SCOREC
6684: C9 0A 836 CMP #$0A
6686: 90 0B 837 BLT SCRSET ;SKIP IF LESS 999
6688: A9 00 838 LDA #$00 ;RESET TO 0 IF 1000
668A: 8D 1E 60 839 STA SCOREA
668D: 8D 1F 60 840 STA SCOREB

```

```

6690: 8D 20 60 841      STA SCOREC
      842 *
      843 *SCORE SETUP ROUTINE FOR DRAW
      844 *
6693: A9 20 845 SCRSET LDA #$20
6695: 85 27 846      STA HIRESH
6697: A9 23 847      LDA #$23      ;SETUP SCREEN LOCATION TO PLOT --
6699: 85 26 848      STA HIRESL    ;SCOREC ,100'S DIGIT
669B: A9 01 849      LDA #$01      ;DIGIT 1 BYTE WIDE
669D: 8D 10 60 850      STA LNGH
66A0: A9 6A 851      LDA #>SCORESH
66A2: 85 51 852      STA SHPH
66A4: AC 20 60 853      LDY SCOREC
66A7: B9 30 6A 854      LDA SCOREP,Y ;INDEX TO CORRECT SHAPE FOR DIGIT--
66AA: 85 50 855      STA SHPL      ;DRAWN
66AC: 20 E8 66 856      JSR SCOREDR ;DRAW 100'S DIGIT
66AF: A9 20 857      LDA #$20      ;SETUP SCREEN LOCATION TO
66B1: 85 27 858      STA HIRESH
66B3: A9 24 859      LDA #$24      ;PLOT SCOREB ,10'S DIGIT
66B5: 85 26 860      STA HIRESL
66B7: A9 01 861      LDA #$01
66B9: 8D 10 60 862      STA LNGH
66BC: A9 6A 863      LDA #>SCORESH
66BE: 85 51 864      STA SHPH
66C0: AC 1F 60 865      LDY SCOREB
66C3: B9 30 6A 866      LDA SCOREP,Y
66C6: 85 50 867      STA SHPL
66C8: 20 E8 66 868      JSR SCOREDR ;DRAW 10'S DIGIT
66CB: A9 20 869      LDA #$20
66CD: 85 27 870      STA HIRESH
66CF: A9 25 871      LDA #$25      ;SETUP SCREEN LOCATION TO
66D1: 85 26 872      STA HIRESL    ;PLOT SCOREA, 1'S DIGIT
66D3: A9 01 873      LDA #$01
66D5: 8D 10 60 874      STA LNGH
66D8: A9 6A 875      LDA #>SCORSH
66DA: 85 51 876      STA SHPH
66DC: AC 1E 60 877      LDY SCOREA
66DF: B9 30 6A 878      LDA SCOREP,Y
66E2: 85 50 879      STA SHPL
66E4: 20 E8 66 880      JSR SCOREDR ;DRAW 1'S DIGIT
66E7: 60      881      RTS
      882 *
      883 *SCORE DRAWING ROUTINE
      884 *
66E8: A2 00 885 SCOREDR LDX #$00
66EA: A0 00 886      LDY #$00      ;OFFSET INTO LINE ALREADY SET --
66EC: A1 50 887 SCORED2 LDA (SHPL,X) ;IN SCRSET
66EE: 91 26 888      STA (HIRESL),Y
66F0: A5 27 889      LDA HIRESH
66F2: 18      890      CLC
66F3: 69 04 891      ADC #$04
66F5: 85 27 892      STA HIRESH
66F7: E6 50 893      INC SHPL
66F9: C9 40 894      CMP #$40
66FB: 90 EF 895      BCC SCORED2
66FD: E9 20 896      SBC #$20
66FF: 85 27 897      STA HIRESH
6701: CE 10 60 898      DEC LNGH
6704: F0 03 899      BEQ SCORED3
6706: C8      900      INY

```



```

6707: DO E3      901      BNE SCORED2
6709: 60          902 SCORED3 RTS
                   903 *
                   904 *T A B L E S *****
                   905 *
                   906 *VERTICAL TABLES

670A: 00 00 00
670D: 00 00 00
6710: 00 00      907 YVERTL  HEX  0000000000000000
6712: 80 80 80
6715: 80 80 80
6718: 80 80      908      HEX  8080808080808080
671A: 00 00 00
671D: 00 00 00
6720: 00 00      909      HEX  0000000000000000
6722: 80 80 80
6725: 80 80 80
6728: 80 80      910      HEX  8080808080808080
672A: 00 00 00
672D: 00 00 00
6730: 00 00      911      HEX  0000000000000000
6732: 80 80 80
6735: 80 80 80
6738: 80 80      912      HEX  8080808080808080
673A: 00 00 00
673D: 00 00 00
6740: 00 00      913      HEX  0000000000000000
6742: 80 80 80
6745: 80 80 80
6748: 80 80      914      HEX  8080808080808080
674A: 28 28 28
674D: 28 28 28
6750: 28 28      915      HEX  2828282828282828
6752: A8 A8 A8
6755: A8 A8 A8
6758: A8 A8      916      HEX  A8A8A8A8A8A8A8A8
675A: 28 28 28
675D: 28 28 28
6760: 28 28      917      HEX  2828282828282828
6762: A8 A8 A8
6765: A8 A8 A8
6768: A8 A8      918      HEX  A8A8A8A8A8A8A8A8
676A: 28 28 28
676D: 28 28 28
6770: 28 28      919      HEX  2828282828282828
6772: A8 A8 A8
6775: A8 A8 A8
6778: A8 A8      920      HEX  A8A8A8A8A8A8A8A8
677A: 28 28 28
677D: 28 28 28
6780: 28 28      921      HEX  2828282828282828
6782: A8 A8 A8
6785: A8 A8 A8
6788: A8 A8      922      HEX  A8A8A8A8A8A8A8A8
678A: 50 50 50
678D: 50 50 50
6790: 50 50      923      HEX  5050505050505050
6792: DO DO DO
6795: DO DO DO
6798: DO DO      924      HEX  DODODODODODODODO

```

679A:	50	50	50			
679D:	50	50	50			
67A0:	50	50		925	HEX	5050505050505050
67A2:	D0	D0	D0			
67A5:	D0	D0	D0			
67A8:	D0	D0		926	HEX	D0D0D0D0D0D0D0D0
67AA:	50	50	50			
67AD:	50	50	50			
67B0:	50	50		927	HEX	5050505050505050
67B2:	D0	D0	D0			
67B5:	D0	D0	D0			
67B8:	D0	D0		928	HEX	D0D0D0D0D0D0D0D0
67BA:	50	50	50			
67BD:	50	50	50			
67C0:	50	50		929	HEX	5050505050505050
67C2:	D0	D0	D0			
67C5:	D0	D0	D0			
67C8:	D0	D0		930	HEX	D0D0D0D0D0D0D0D0
				931	*	
67CA:	20	24	28			
67CD:	2C	30	34			
67D0:	38	3C		932	YVERTH	HEX 2024282C3034383C
67D2:	20	24	28			
67D5:	2C	30	34			
67D8:	38	3C		933	HEX	2024282C3034383C
67DA:	21	25	29			
67DD:	2D	31	35			
67E0:	39	3D		934	HEX	2125292D3135393D
67E2:	21	25	29			
67E5:	2D	31	35			
67E8:	39	3D		935	HEX	2125292D3135393D
67EA:	22	26	2A			
67ED:	2E	32	36			
67F0:	3A	3E		936	HEX	22262A2E32363A3E
67F2:	22	26	2A			
67F5:	2E	32	36			
67F8:	3A	3E		937	HEX	22262A2E32363A3E
67FA:	23	27	2B			
67FD:	2F	33	37			
6800:	3B	3F		938	HEX	23272B2F33373B3F
6802:	23	27	2B			
6805:	2F	33	37			
6808:	3B	3F		939	HEX	23272B2F33373B3F
680A:	20	24	28			
680D:	2C	30	34			
6810:	38	3C		940	HEX	2024282C3034383C
6812:	20	24	28			
6815:	2C	30	34			
6818:	38	3C		941	HEX	2024282C3034383C
681A:	21	25	29			
681D:	2D	31	35			
6820:	39	3D		942	HEX	2125292D3135393D
6822:	21	25	29			
6825:	2D	31	35			
6828:	39	3D		943	HEX	2125292D3135393D
682A:	22	26	2A			
682D:	2E	32	36			
6830:	3A	3E		944	HEX	22262A2E32363A3E
6832:	22	26	2A			
6835:	2E	32	36			

6838:	3A 3E	945	HEX	22262A2E32363A3E
683A:	23 27 2B			
683D:	2F 33 37			
6840:	3B 3F	946	HEX	23272B2F33373B3F
6842:	23 27 2B			
6845:	2F 33 37			
6848:	3B 3F	947	HEX	23272B2F33373B3F
684A:	20 24 28			
684D:	2C 30 34			
6850:	38 3C	948	HEX	2024282C3034383C
6852:	20 24 28			
6855:	2C 30 34			
6858:	38 3C	949	HEX	2024282C3034383C
685A:	21 25 29			
685D:	2D 31 35			
6860:	39 3D	950	HEX	2125292D3135393D
6862:	21 25 29			
6865:	2D 31 35			
6868:	39 3D	951	HEX	2125292D3135393D
686A:	22 26 2A			
686D:	2E 32 36			
6870:	3A 3E	952	HEX	22262A2E32363A3E
6872:	22 26 2A			
6875:	2E 32 36			
6878:	3A 3E	953	HEX	22262A2E32363A3E
687A:	23 27 2B			
687D:	2F 33 37			
6880:	3B 3F	954	HEX	23272B2F33373B3F
6882:	23 27 2B			
6885:	2F 33 37			
6888:	3B 3F	955	HEX	23272B2F33373B3F
		956	*	
		957	*TABLES TO KEEP TRACK OF OBJECTS	
		958	*	
688A:	00 00 00			
688D:	00 00 00			
6890:	00	959	TABLEX	HEX 00000000000000
6891:	28 38 48			
6894:	58 68 28			
6897:	38	960	TABLEY	HEX 28384858682838
6898:	01 01 01			
689B:	01 01 01			
689E:	01	961	ALIVE	HEX 01010101010101
689F:	00 00 00			
68A2:	00 00 00			
68A5:	00	962	USFLAG	HEX 00000000000000
68A6:	00 00 00			
68A9:	00 00 00			
68AC:	00	963	ONFLAG	HEX 00000000000000
68AD:	2D 40 70			
68B0:	90 C0 D0			
68B3:	F0	964	ONPOS	HEX 2D407090C0D0F0
68B4:	00 00 01			
68B7:	00 00 00			
68BA:	01	965	SHPADR	HEX 00000100000001
		966	*	
68BB:	04	967	SHPLO	DFB SHAPES
68BC:	14	968	DFB	SHAPES+\$10
68BD:	24	969	DFB	SHAPES+\$20
68BE:	34	970	DFB	SHAPES+\$30

```

971 *
972 *MASK SHIP TABLE
68BF: 01 00 00
68C2: 03 00 00
68C5: 07 00 973 MSHIP      HEX  0100000300000700
68C7: 00 0F 00
68CA: 00 7F 7F
68CD: 00 7F 974          HEX  000F00007F7F007F
68CF: 1F 07 7F
68D2: 7F 1F 78
68D5: 7F 7F 975          HEX  1F077F7F1F787F7F
976 *SHAPE TABLE SHIP
68D7: 80 00 00
68DA: 82 00 00
68DD: 82 00 977 SHIP      HEX  8000008200008200
68DF: 00 8A 00
68E2: 00 AA D5
68E5: 80 AA 978          HEX  008A0000AAD580AA
68E7: 95 82 AA
68EA: D5 8A A8
68ED: D5 AA 979          HEX  9582AAD58AA8D5AA
980 *
981 *SHAPE BOMB
68EF: 07 7E 07 982 SHBOMB   HEX  077E07
983          DS  18
984 *
985 *SHAPE ALIEN EVEN
6904: 28 28 0A
6907: 2A 2A 22
690A: 22 22 986 SHAPES    HEX  28280A2A2A222222
690C: 00 01 01
690F: 01 05 04
6912: 04 04 987          HEX  0001010105040404
988 *SHAPE SAUCER EVEN
6914: 40 70 30
6917: AA AA 70
691A: 00 00 989          HEX  407030AAAA700000
691C: 01 07 06
691F: D5 D5 07
6922: 00 00 990          HEX  010706D5D5070000
991 *ODD ALIEN SHAPE
6924: 50 54 04
6927: 54 55 11
692A: 11 11 992          HEX  5054045455111111
692C: 00 00 02
692F: 02 02 02
6932: 02 02 993          HEX  0000020202020202
994 *ODD SAUCER SHAPE
6934: 40 70 30
6937: D5 D5 70
693A: 00 00 995          HEX  407030D5D5700000
693C: 01 07 06
693F: AA AA 07
6942: 00 00 996          HEX  010706AAAA070000
997 *
998 *EXPLOSION TABLES
6944: 08 09 0A
6947: 0B 0B 0A
694A: 09 08 999 EOFFX     HEX  08090A0B0B0A0908
694C: 07 08 09

```

694F: 0A 0B 0C		
6952: 0C 0B	1000	HEX 0708090A0B0C0C0B
6954: 0A 08 07		
6957: 05 06 08		
695A: 09 0A	1001	HEX 0A0807050608090A
695C: 0C 0D 0E		
695F: 0E 0D 0C		
6962: 0B 09	1002	HEX 0C0D0E0E0D0C0B09
6964: 07 06 04		
6967: 05 06 08		
696A: 0A 0C	1003	HEX 0706040506080A0C
696C: 0E 0F 0F		
696F: 0E 0D 0B		
6972: 09 07	1004	HEX 0E0F0F0E0D0B0907
6974: 05 04 02		
6977: 03 05 08		
697A: 0B 0D	1005	HEX 0504020305080B0D
697C: 0F 10 11		
697F: 10 0F 0D		
6982: 0B 08	1006	HEX 0F1011100F0D0B08
6984: 06 04 03		
6987: 02 00 01		
698A: 04 07	1007	HEX 0604030200010407
698C: 0A 0E 11		
698F: 12 13 12		
6992: 11 0F	1008	HEX 0A0E11121312110F
6994: 0B 07 04		
6997: 02 01 00	1009	HEX 0B0704020100
699A: FC F8 F8		
699D: FC 04 08		
69A0: 08 04	1010 EOFFY	HEX FCF8F8FC04080804
69A2: F8 F0 EC		
69A5: EC F0 F8		
69A8: 04 0C	1011	HEX F8F0ECECF0F8040C
69AA: 10 0C 04		
69AD: F8 EC E4		
69B0: E0 E4	1012	HEX 100C04F8ECE4E0E4
69B2: E4 EC F4		
69B5: 00 0C 14		
69B8: 18 1C	1013	HEX E4ECF4000C14181C
69BA: 14 08 F0		
69BD: E4 DC D4		
69C0: D4 DC	1014	HEX 1408F0E4DCD4D4DC
69C2: E4 F0 00		
69C5: 14 20 24		
69C8: 28 20	1015	HEX E4F0001420242820
69CA: 14 00 EC		
69CD: E0 D4 CC		
69D0: C8 D0	1016	HEX 1400ECE0D4CCC8D0
69D2: D8 E8 FC		
69D5: 14 24 2C		
69D8: 34 34	1017	HEX D8E8FC14242C3434
69DA: 2C 20 10		
69DD: 00 E4 D0		
69E0: C8 C0	1018	HEX 2C201000E4D0C8C0
69E2: B8 C4 D4		
69E5: E4 FC 18		
69E8: 2C 38	1019	HEX B8C4D4E4FC182C38
69EA: 48 40 38		
69ED: 28 10 00	1020	HEX 484038281000

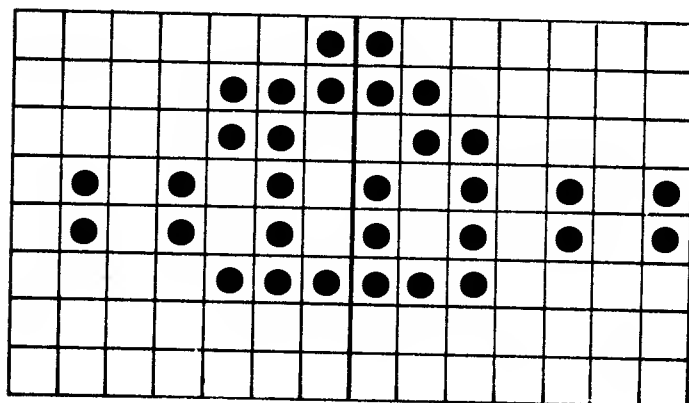
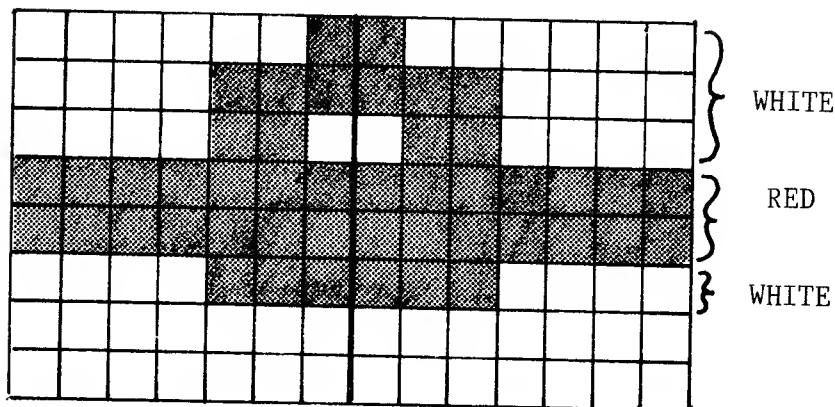
	1021	DS	24
	1022 *		
	1023	*SHAPES FOR SCOREKEEPING	
6A08: 3F 01 01			
6A0B: 3F 20 20			
6A0E: 3F 00	1024	SCOREWD	HEX 3F01013F20203F00
6A10: 3C 02 01			
6A13: 01 01 02			
6A16: 3C 00	1025		HEX 3C02010101023C00
6A18: 1E 21 21			
6A1B: 21 21 21			
6A1E: 1E 00	1026		HEX 1E21212121211E00
6A20: 3F 21 21			
6A23: 3F 09 11			
6A26: 21 00	1027		HEX 3F21213F09112100
6A28: 3F 01 01			
6A2B: 1F 01 01			
6A2E: 3F 00	1028		HEX 3F01011F01013F00
	1029	*INDEX TO LO BYTE SCORE NUMBER SHAPES	
6A30: 3A	1030	SCOREP	DFB SCORESH
6A31: 42	1031		DFB SCORESH+\$08
6A32: 4A	1032		DFB SCORESH+\$10
6A33: 52	1033		DFB SCORESH+\$18
6A34: 5A	1034		DFB SCORESH+\$20
6A35: 62	1035		DFB SCORESH+\$28
6A36: 6A	1036		DFB SCORESH+\$30
6A37: 72	1037		DFB SCORESH+\$38
6A38: 7A	1038		DFB SCORESH+\$40
6A39: 82	1039		DFB SCORESH+\$48
	1040 *		
	1041	*NUMBER SHAPES	
6A3A: 1C 22 22			
6A3D: 22 22 22			
6A40: 1C 00	1042	SCORESH	HEX 1C22222222221C00
6A42: 08 0C 08			
6A45: 08 08 08			
6A48: 1C 00	1043		HEX 080C080808081C00
6A4A: 1C 22 20			
6A4D: 18 04 02			
6A50: 3E 00	1044		HEX 1C22201804023E00
6A52: 3E 20 10			
6A55: 08 10 22			
6A58: 1C 00	1045		HEX 3E20100810221C00
6A5A: 18 14 12			
6A5D: 11 3F 10			
6A60: 10 00	1046		HEX 181412113F101000
6A62: 3E 02 02			
6A65: 3E 20 22			
6A68: 1C 00	1047		HEX 3E02023E20221C00
6A6A: 38 04 02			
6A6D: 1E 22 22			
6A70: 1C 00	1048		HEX 3804021E22221C00
6A72: 3E 20 10			
6A75: 08 04 04			
6A78: 04 00	1049		HEX 3E20100804040400
6A7A: 1C 22 22			
6A7D: 1C 22 22			
6A80: 1C 00	1050		HEX 1C22221C22221C00
6A82: 1C 22 22			
6A85: 1E 20 10			

```

6A88: 0E 00    1051      HEX  1C22221E20100E00
6A8A: 1C 22 22
6A8D: 22 22 22
6A90: 1C 00    1052      HEX  1C2222222221C00

```

--END ASSEMBLY-- 2706 BYTES



EVEN		ODD	
40	01	40	01
70	07	70	07
30	06	30	06
AA	D5	D5	AA
AA	D5	D5	AA
70	07	70	07
00	00	00	00
00	00	00	00

B R B R B R B R B R B R B R

EVEN OFFSET SHAPE

HI-RES SCREEN SCROLLING

There are an increasing number of games that require fast scrolling. Racing car games, where the screen (or at least sections of the screen scroll) rapidly vertically, are good examples. It is certainly much easier to scroll the screen in

that direction, because only two adjacent lines are involved, and the screen addresses for those two lines are easily referenced from lookup tables.

The algorithm for scrolling down the screen involves taking the bytes from one line and storing them in the line directly below. This is done across a row for each column. The most important thing is that you start from the bottom of the screen or you will overwrite lines. Also, the bottom line must be transferred to the top of the screen if a wrap-a-round effect is desired. A cute trick which minimizes the code considerably is to extend the YVERT table one extra byte. That byte is the address of the 0th line. Therefore, line #191 can be moved to line #192, which is actually line #0.

Moving an entire screen upwards a single line by this method is not that fast, but usually, as in racing games, only narrow background strips need to be scrolled. This produces more reasonable scrolling rates. Other techniques involve using a background that occupies every other screen line, then scrolling it two lines at a time. The Phantom's Five game appears to use this method. Another approach is to utilize straight in-line code, where scrolling for all the lines is done a column at a time. Bytes are moved upwards with the following code

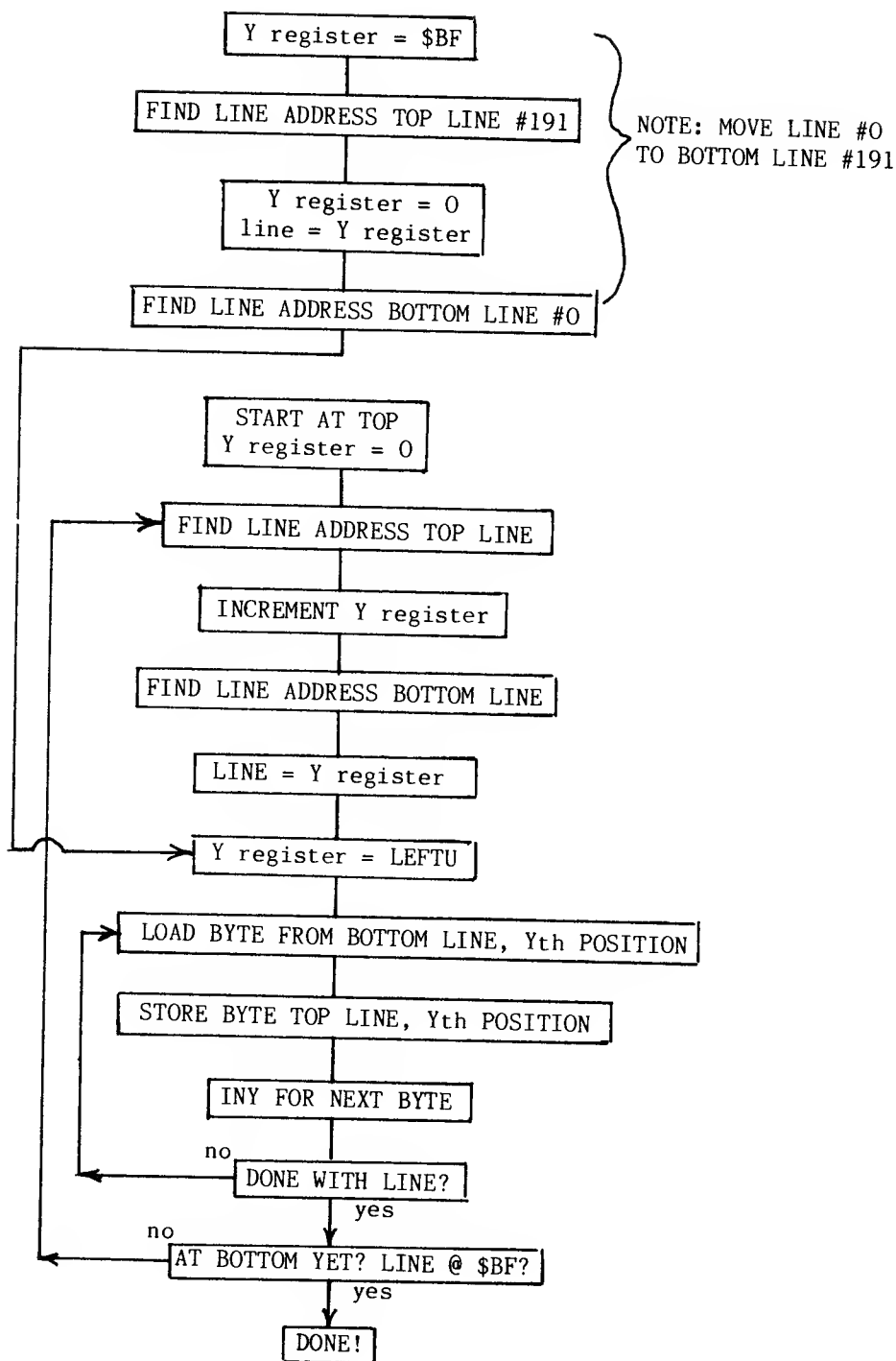
```
LDA $3C00,Y
STA $3F00,Y
.      .      .
.      .      .
LDA $2800,Y
STA $2C00,Y
LDA $2400,Y
STA $2800,Y
LDA $2000,Y
STA $2400,Y
```

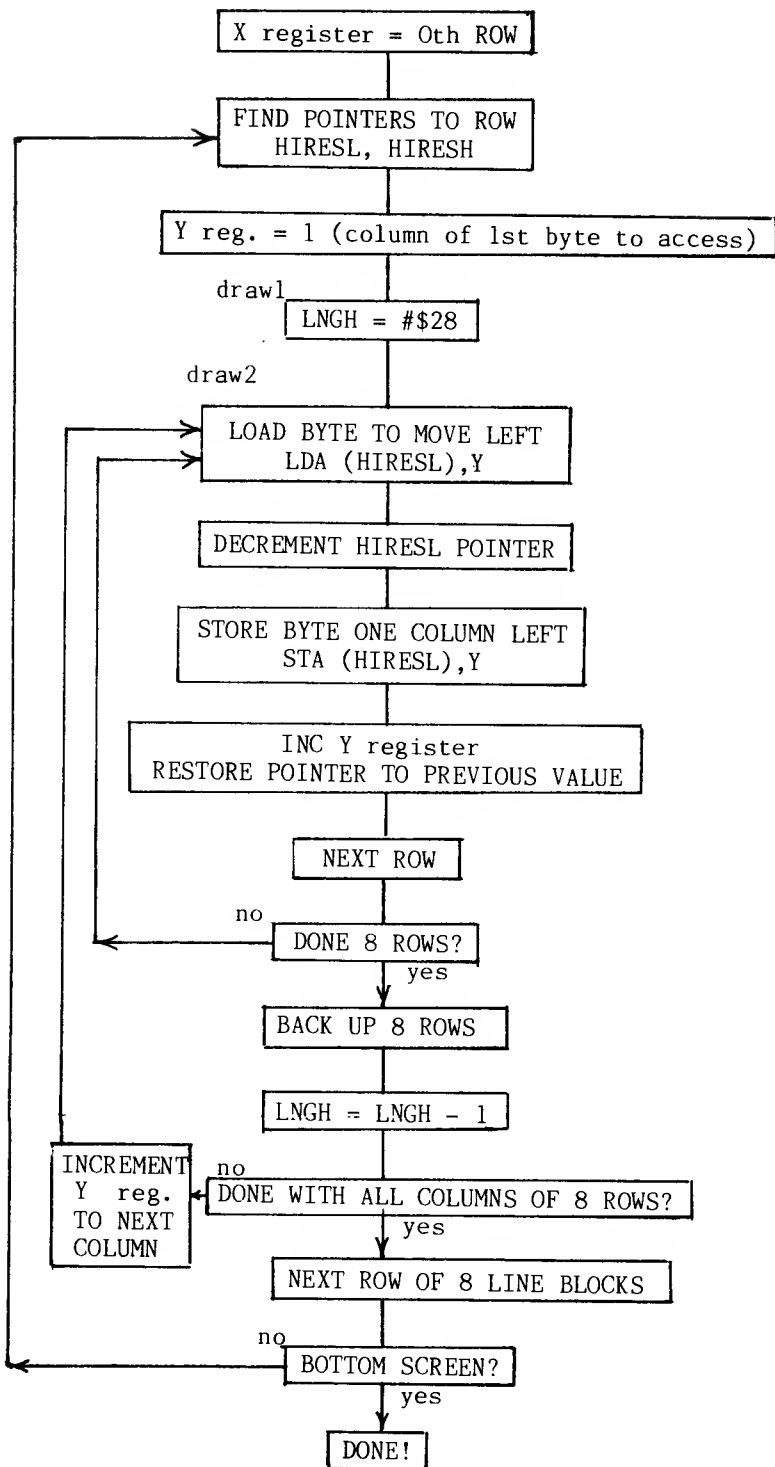
where Y is looped from \$0 to \$27 across the screen. This code is at least three times faster than the first method.

Scrolling the screen upwards is quite similar to scrolling the screen downwards. It requires moving the screen memory from the lower line to the upper line, across all 40 columns. The bytes in the 0th line must be moved to the 191st line if a wrap-a-round effect is desired. This requires extra code, since we can't do any fancy tricks as we did before.

The two scrolling routines, one up and one down, have been put together in the following program. The scrolling windows have been set so that part of the screen scrolls up and part of the screen scrolls down, while the remainder remains stationary. The variables that control the windows are LEFT and RIGHT for scrolling down, and LEFTU and RIGHTU for scrolling up. These values can be modified in lines 16, 18, 20 and 22.

The flow charts and code are presented below:





```

1  *SCROLL UP & DOWN SUBROUTINE
2      ORG $6000
6000: 4C 08 60 3  JMP PROG
4      LEFT DS 1
5      RIGHT DS 1
6      LINE DS 1
7      LEFTU DS 1
8      RIGHTU DS 1
9      TOPL EQU $6
10     TOPH EQU TOPL+$1
11     BOTTOML EQU $8
12     BOTTOMH EQU BOTTOML+$1
6008: AD 50 C0 13  LDA $C050
600B: AD 52 C0 14  LDA $C052
600E: AD 57 C0 15  LDA $C057
6011: A9 06 16  LDA #$06
6013: 8D 03 60 17  STA LEFT ;LEFT WINDOW SCROLL DOWN
6016: A9 0A 18  LDA #$0A
6018: 8D 04 60 19  STA RIGHT ;RIGHT WINDOW SCROLL DOWN
601B: A9 20 20  LDA #$20
601D: 8D 06 60 21  STA LEFTU ;LEFT WINDOW SCROLL UP
6020: A9 25 22  LDA #$25
6022: 8D 07 60 23  STA RIGHTU ;RIGHT WINDOW SCROLL UP
6025: 20 2E 60 24  CONT JSR SCROLL
6028: 20 5D 60 25  JSR SCROLLU
602B: 4C 25 60 26  JMP CONT
27  *SCROLL DOWN SUBROUTINE
602E: A0 C0 28  SCROLL LDY #$C0 ;START WITH BOTTOM LINE --
29  * ;AND WORK TO TOP
6030: B9 AA 60 30  START LDA YVERTL,Y ;FIND SCREEN ADDRESS --
6033: 85 08 31  STA BOTTOML ;OF BOTTOM LINE
6035: B9 6B 61 32  LDA YVERTH,Y
6038: 85 09 33  STA BOTTOMH
603A: 88 34  DEY ;DECREMENT LINE NUMBER
603B: B9 AA 60 35  LDA YVERTL,Y ;FIND SCREEN ADDRESS TOP LINE
603E: 85 06 36  STA TOPL
6040: B9 6B 61 37  LDA YVERTH,Y
6043: 85 07 38  STA TOPH
6045: 8C 05 60 39  STY LINE ;TEMP STORE Y REGISTER
6048: AC 03 60 40  LDY LEFT ;START SHIFTING LINE
604B: B1 06 41  LOOP LDA (TOPL),Y ;LOAD BYTE ON SCREEN
604D: 91 08 42  STA (BOTTOML),Y ;STORE BYTE ON LINE BELOW
604F: C8 43  INY ;NEXT BYTE
6050: CC 04 60 44  CPY RIGHT ;DONE WITH LINE?
6053: D0 F6 45  BNE LOO ;NO,DO NEXT BYTE ON LINE
6055: AC 05 60 46  LDY LINE ;RESET Y REGISTER WITH LINE
6058: C0 00 47  CPY #$00 ;AT TOP YET?
605A: D0 D4 48  BNE START
605C: 60 49  RTS
50  *SCROLL UP SUBROUTINE
51  *FIRST TAKE TOP LINE AND PUT ON BOTTOM
52  *IN THIS SPECIAL CASE THINK OF IT AS LINE #0 BELOW LINE #191
605D: A0 BF 53  SCROLLU LDY #$BF ;LINE #191
605F: B9 AA 60 54  LDA YVERTL,Y ;FIND SCREEN ADDRESS --
6062: 85 06 55  STA TOPL ;OF TOP LINE
6064: B9 6B 61 56  LDA YVERTH,Y
6067: 85 07 57  STA TOPH
6069: A0 00 58  LDY #$00
606B: 8C 05 60 59  STY LINE
606E: B9 AA 60 60  LDA YVERTL,Y ;FIND SCREEN ADDRESS --

```

6071: 85 08 61		STA BOTTOML	;OF BOTTOM LINE
6073: B9 6B 61 62		LDA YVERTH,Y	
6076: 85 09 63		STA BOTTOMH	
6078: 4C 95 60 64		JMP LOOP2-3	;GOTO INSTRUCTION BEFORE LOOP2
607B: A0 00 65		LDY #\$00	;START AT TOP
607D: B9 AA 60 66	STARTU	LDA YVERTL,Y	;FIND SCREEN ADDRESS --
6080: 85 06 67		STA TOPL	;OF TOP LINE
6082: B9 6B 61 68		LDA YVERTH,Y	
6085: 85 07 69		STA TOPH	
6087: C8 70		INY	;NEXT ROW
6088: B9 AA 60 71		LDA YVERTL,Y	;FIND SCREEN ADDRESS --
608B: 85 08 72		STA BOTTOML	;OF BOTTOM LINE
608D: B9 6B 61 73		LDA YVERTH,Y	
6090: 85 09 74		STA BOTTOMH	
6092: 8C 05 60 75		STY LINE	;TEMP STORE Y REGISTER
6095: AC 06 60 76		LDY LEFTU	;START SHIFTING LINE
6098: B1 08 77	LOOP2	LDA (BOTTOML),Y	;LOAD BYTE ON SCREEN
609A: 91 06 78		STA (TOPL),Y	;STORE BYTE ON LINE ABOVE
609C: C8 79		INY	;NEXT BYTE
609D: CC 07 60 80		CPY RIGHTU	;DONE WITH LINE?
60A0: D0 F6 81		BE LOOP2	;NO,DO NEXT BYTE ON LINE
60A2: AC 05 60 82		LDY LINE	;RESET Y REG. WITH LINE
60A5: C0 BF 83		CPY #\$BF	;AT BOTTOM YET?
60A7: D0 D4 84		BNE STARTU	
60A9: 60 85		RTS	
60AA: 00 00 00			
60AD: 00 00 00			
60B0: 00 00 86	YVERTL	HEX	0000000000000000
60B2: 80 80 80			
60B5: 80 80 80			
60B8: 80 80 87		HEX	8080808080808080
60BA: 00 00 00			
60BD: 00 00 00			
60C0: 00 00 88		HEX	0000000000000000
60C2: 80 80 80			
60C5: 80 80 80			
60C8: 80 80 89		HEX	8080808080808080
60CA: 00 00 00			
60CD: 00 00 00			
60D0: 00 00 90		HEX	0000000000000000
60D2: 80 80 80			
60D5: 80 80 80			
60D8: 80 80 91		HEX	8080808080808080
60DA: 00 00 00			
60DD: 00 00 00			
60E0: 00 00 92		HEX	0000000000000000
60E2: 80 80 80			
60E5: 80 80 80			
60E8: 80 80 93		HEX	8080808080808080
60EA: 28 28 28			
60ED: 28 28 28			
60F0: 28 28 94		HEX	2828282828282828
60F2: A8 A8 A8			
60F5: A8 A8 A8			
60F8: A8 A8 95		HEX	A8A8A8A8A8A8A8A8
60FA: 28 28 28			
60FD: 28 28 28			
6100: 28 28 96		HEX	2828282828282828
6102: A8 A8 A8			
6105: A8 A8 A8			

6108: A8 A8	97		HEX	A8A8A8A8A8A8A8A8
610A: 28 28 28				
610D: 28 28 28				
6110: 28 28	98		HEX	2828282828282828
6112: A8 A8 A8				
6115: A8 A8 A8				
6118: A8 A8	99		HEX	A8A8A8A8A8A8A8A8
611A: 28 28 28				
611D: 28 28 28				
6120: 28 28	100		HEX	2828282828282828
6122: A8 A8 A8				
6125: A8 A8 A8				
6128: A8 A8	101		HEX	A8A8A8A8A8A8A8A8
612A: 50 50 50				
612D: 50 50 50				
6130: 50 50	102		HEX	5050505050505050
6132: D0 D0 D0				
6135: D0 D0 D0				
6138: D0 D0	103		HEX	D0D0D0D0D0D0D0D0
613A: 50 50 50				
613D: 50 50 50				
6140: 50 50	104		HEX	5050505050505050
6142: D0 D0 D0				
6145: D0 D0 D0				
6148: D0 D0	105		HEX	D0D0D0D0D0D0D0D0
614A: 50 50 50				
614D: 50 50 50				
6150: 50 50	106		HEX	5050505050505050
6152: D0 D0 D0				
6155: D0 D0 D0				
6158: D0 D0	107		HEX	D0D0D0D0D0D0D0D0
615A: 50 50 50				
615D: 50 50 50				
6160: 50 50	108		HEX	5050505050505050
6162: D0 D0 D0				
6165: D0 D0 D0				
6168: D0 D0 00	109		HEX	D0D0D0D0D0D0D0D000
	110	*		
616B: 20 24 28				
616E: 2C 30 34				
6171: 38 3C	111	YVERTH	HEX	2024282C3034383C
6173: 20 24 28				
6176: 2C 30 34				
6179: 38 3C	112		HEX	2024282C3034383C
617B: 21 25 29				
617E: 2D 31 35				
6181: 39 3D	113		HEX	2125292D3135393D
6183: 21 25 29				
6186: 2D 31 35				
6189: 39 3D	114		HEX	2125292D3135393D
618B: 22 26 2A				
618E: 2E 32 36				
6191: 3A 3E	115		HEX	22262A2E32363A3E
6193: 22 26 2A				
6196: 2E 32 36				
6199: 3A 3E	116		HEX	22262A2E32363A3E
619B: 23 27 2B				
619E: 2F 33 37				
61A1: 3B 3F	117		HEX	23272B2F33373B3F
61A3: 23 27 2B				

61A6: 2F 33 37		
61A9: 3B 3F	118	HEX 23272B2F33373B3F
61AB: 20 24 28		
61AE: 2C 30 34		
61B1: 38 3C	119	HEX 2024282C3034383C
61B3: 20 24 28		
61B6: 2C 30 34		
61B9: 38 3C	120	HEX 2024282C3034383C
61BB: 21 25 29		
61BE: 2D 31 35		
61C1: 39 3D	121	HEX 2125292D3135393D
61C3: 21 25 29		
61C6: 2D 31 35		
61C9: 39 3D	122	HEX 2125292D3135393D
61CB: 22 26 2A		
61CE: 2E 32 36		
61D1: 3A 3E	123	HEX 22262A2E32363A3E
61D3: 22 26 2A		
61D6: 2E 32 36		
61D9: 3A 3E	124	HEX 22262A2E32363A3E
61DB: 23 27 2B		
61DE: 2F 33 37		
61E1: 3B 3F	125	HEX 23272B2F33373B3F
61E3: 23 27 2B		
61E6: 2F 33 37		
61E9: 3B 3F	126	HEX 23272B2F33373B3F
61EB: 20 24 28		
61EE: 2C 30 34		
61F1: 38 3C	127	HEX 2024282C3034383C
61F3: 20 24 28		
61F6: 2C 30 34		
61F9: 38 3C	128	HEX 2024282C3034383C
61FB: 21 25 29		
61FE: 2D 31 35		
6201: 39 3D	129	HEX 2125292D3135393D
6203: 21 25 29		
6206: 2D 31 35		
6209: 39 3D	130	HEX 2125292D3135393D
620B: 22 26 2A		
620E: 2E 32 36		
6211: 3A 3E	131	HEX 22262A2E32363A3E
6213: 22 26 2A		
6216: 2E 32 36		
6219: 3A 3E	132	HEX 22262A2E32363A3E
621B: 23 27 2B		
621E: 2F 33 37		
6221: 3B 3F	133	HEX 23272B2F33373B3F
6223: 23 27 2B		
6226: 2F 33 37		
6229: 3B 3F 20	134	HEX 23272B2F33373B3F20

--END ASSEMBLY--

ERRORS: 0

556 BYTES

Scrolling the screen left or right in the horizontal direction is slightly more difficult. The normal scrolling direction for games is left, because objects in most games travel from left to right, and the background terrain scrolls left. This method moves each byte in one of the 8 line subgroups leftwards, a byte at a time. Byte-shifting starts at the 1st column, moving that byte to the 0th column, then drops down to the next row, moves a byte again, until all eight rows have been moved. Then the routine increments the column number and repeats the operation until all 40 columns of eight rows have been moved. It does this for all 24 subgroups.

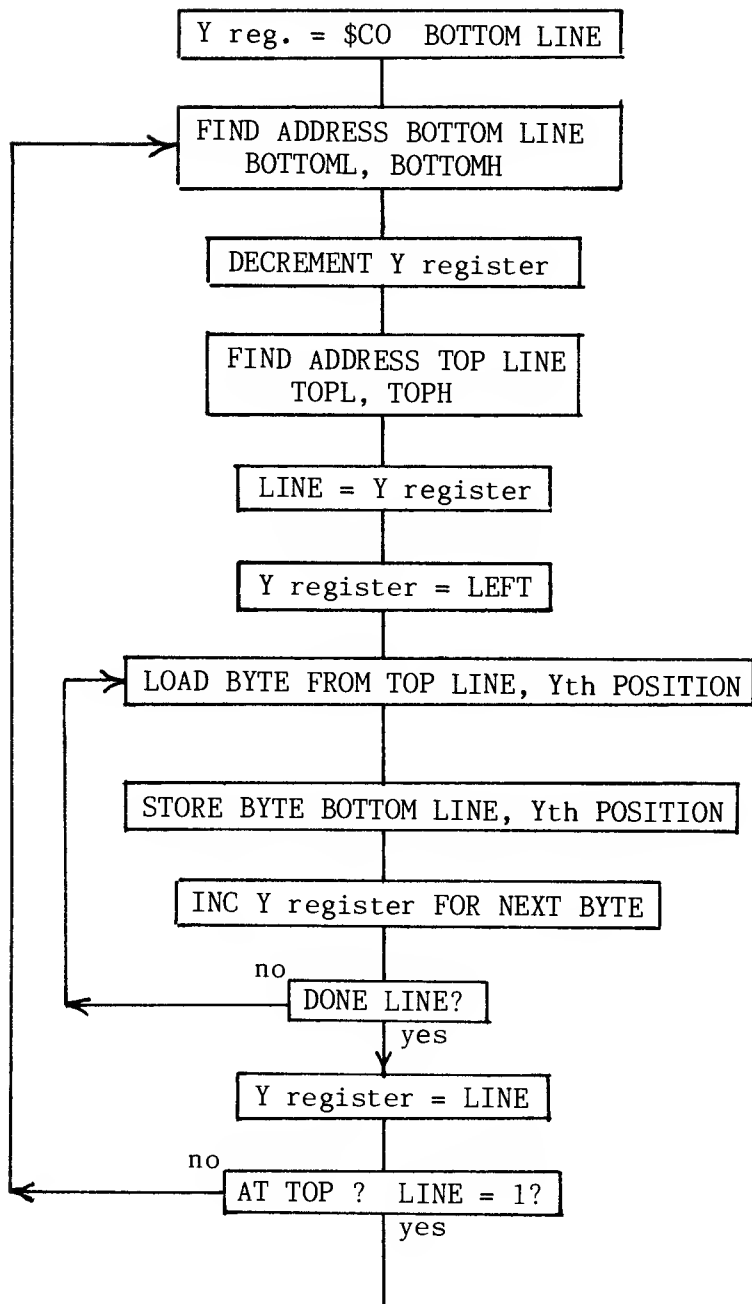
Normally, during scrolling, a new column of data is plotted at the 39th column. Wrap-a-round is tricky, because when a byte is moved off the screen's left side it will reappear on a line $\frac{1}{2}$ higher on the screen. If you would like to see this strange scrolling effect, change the value in line #25 to #28.

Both the code and flow chart are shown below.

```

1      *SCROLL LEFT SUBROUTINE
2      ORG $6000
6000: 4C 05 60 3      JMP PROG
4      BLOCK DS 1
5      LNGH DS 1
6      HIRESL EQU $FB
7      HIRESH EQU HIRESL+$1
8      *ENTER HERE FIRST TIME ACCESS
6005: AD 50 C0 9      PROG LDA $C050
6008: AD 52 C0 10     LDA $C052
600B: AD 57 C0 11     LDA $C057
600E: A2 00 12      START LDX #$00 ;OTH ROW OF 8 LINE BLOCKS
6010: BD 4A 60 13     NXBLOCK LDA YBLOCKH,X ;GET SCREEN POINTERS FOR 1ST ROW -
6013: 85 FC 14       STA HIRESH ;OF BLOCK
6015: BD 62 60 15     LDA YBLOCKL,X
6018: 85 FB 16       STA HIRESL
601A: A0 01 17       LDY #$01 ;NEED TO MOVE COLUMN #1 BYTE FIRST
601C: 20 27 60 18     JSR DRAW1
601F: E8 19         INX ;NEXT ROW
6020: E0 18 20       CPX #$18 ;BOTTOM YET?
6022: 90 EC 21       BLT NXBLOCK ;NO, CONTINUE
6024: 4C 0E 60 22     JMP START ;SCROLL ENTIRE SCREEN AGAIN
23     *SUBROUTINE TO DRAW EACH SHAPE
24     *EACH SHAPE 1 BYTE BY 8 ROWS
6027: A9 27 25       DRAW1 LDA #$27
6029: 8D 04 60 26     STA LNGH
602C: B1 FB 27       DRAW2 LDA (HIRESL),Y ;LOAD BYTE WANT TO MOVE LEFT
602E: 88 28         DEY ;LO BYTE POINTER TO ONE BYTE LEFT
602F: 91 FB 29       STA (HIRESL),Y ; STORE BYTE
6031: C8 30         INY ;RETURN POINTER TO RIGHT
6032: A5 FC 31       LDA HIRESH
6034: 18 32         CLC
6035: 69 04 33       ADC #$04 ;THIS GETS TO NEXT ROW IN BLOCK
6037: 85 FC 34       STA HIRESH
6039: C9 40 35       CMP #$40 ;ARE WE FINISHED WITH 8 ROWS
603B: 90 EF 36       BCC DRAW2 ;NO DO NEXT BYTE
603D: E9 20 37       SBC #$20 ;RETURN TO TOP ROW
603F: 85 FC 38       STA HIRESH
6041: CE 04 60 39     DEC LNGH

```




```

6044: FO 03      40          BEQ  DRAW3      ;FINISHED?
6046: C8         41          INY              ;NEXT COLUMN OF 8 ROWS
6047: D0 E3      42          BNE  DRAW2
6049: 60         43          DRAW3    RTS
        44          *TABLES OF STARTING VALUE OF EACH OF 20 BLOCKS
604A: 20 20 21
604D: 21 22 22
6050: 23 23 20
6053: 20         45          YBLOCKH  HEX  20202121222223232020
6054: 21 21 22
6057: 22 23 23
605A: 20 20 21
605D: 21         46          HEX  21212222232320202121
605E: 22 22 23
6061: 23         47          HEX  22222323
6062: 00 80 00
6065: 80 00 80
6068: 00 80 28
606B: A8         48          YBLOCKL  HEX  008000800080008028A8
606C: 28 A8 28
606F: A8 28 A8
6072: 50 D0 50
6075: D0         49          HEX  28A828A828A850D050D0
6076: 50 D0 50
6079: D0         50          HEX  50D050D0

```

--END ASSEMBLY--

ERRORS: 0

122 BYTES

WHAT MAKES A GOOD GAME

There is no sure-fire way to predict whether a game will be successful, but there are certain attributes that may ensure success. Certainly, a game should have a goal, for, without one, what is the point in playing? The game should also be challenging, since, without requiring some skill, you would tire of it quickly. A game should evoke either a fantasy situation or your innate curiosity, for, without being novel or puzzling, it becomes boring. And lastly (especially in arcade games), a game should be easily controllable in regards to the interaction of the player with the computer game.

Game objectives take two different forms. There are games where the goal is approached, like destroying the fleet of invaders in *Galaxian* or *Space Invaders*, or landing on the moon in *Lunar lander*. There are also games where the goal is to avoid catastrophe. Examples of this range from preventing a nuclear power plant meltdown in *Three Mile Island* to saving your cities during a nuclear missile attack in *Missile Command*.

Goals must suit a player's expectations or fantasies. This is why certain people like certain types of games more than others. The battle-lines of good against evil lurk in the background of many space games, wherein evil, menacing invaders are bent on destruction of the Earth. It becomes the player's goal to protect the Earth as long as possible while scoring the most points for killing aliens. The fantasy of destroying objects during a game appeals to others. It can take the form of popping balloons by bouncing a clown off a teeter-totter, such as in *Clowns and Balloons*, or breaking out bricks in a wall, as in *Breakout*. In each case, the partially-destroyed wall or rows of balloons presents a visually compelling goal and a graphic scorekeeping device as well. Other goals that appeal to many range from accumulating the most treasure while exploring an underground cavern to escaping from a crumbling building before it collapses or before your food runs out.

Goals in most games imply that there is some end point, either when the goal is reached or when you fail. It is often important to make sure the game doesn't just go on and on forever. Limits should be set. Sometimes these take the form of time limits or the amount of ammunition, balls or ships left.

For a game to be considered challenging, it should have a goal where the outcome is uncertain. If the player is certain to reach the goal or certain not to reach it, the game is unlikely to be a challenge and the player will lose interest. It is very easy to introduce randomness into a game by either hiding important information or introducing random variables that draw the player towards disaster. But you must be careful not to overdo this, since a totally random

game lacks a skill factor. Players quickly discover that they have no control over the outcome.

A variable difficulty level is often used to alter the game's level of play. These levels, often with ego satisfying names like Star Commander or Pilot, can be set by the player. Many games are designed to become harder the further you get into them. This increasing skill level requirement presents an added challenge, while preventing the player from growing complacent. Often, the technique is to speed up the game or place additional enemy craft into the battle. The player is required to play faster and better, honing his reflexes during the process.

Any good game should offer a reward for reaching increasingly difficult levels of play. Often, bonus points, extra balls, ships, or more ammunition are rewarded for exceeding score thresholds. It is important that there be greater rewards for winning than losing. A person's ego is involved. A player wants to beat a challenging game, not to be humiliated each time he loses.

Games either need to fulfill a player's fantasy or stimulate their curiosity. Computer game fantasies derive some of their appeal from the emotional needs that they satisfy. Different fantasies appeal to different people.

Appealing to a player's curiosity is often effective in keeping a game interesting. While novelty is sometimes a crucial factor in the original purchase, if the game has little depth, it becomes repetitious and boring. One method that appeals to many game designers is to have the game progress to slightly different scenarios. Some games change the opposition, while others vary the scenery; some do both. The player has to excel if he is to satisfy his curiosity. Games like Threshold, which progresses through 24 sets of alien spacecraft, or Pegasus II, in which the scenery changes and the attacking aliens vary, offer strong curiosity incentives.

A game's controllability is one of the more important considerations in a game's design. It is sometimes referred to as human engineering. Designer's usually choose between keyboard and paddle/joystick control. While eye/hand coordination is more effective using paddles or joysticks, programmers attempting to create games with too many control functions will opt for a keyboard control system. At times, they produce a game that requires nine or ten keyboard controls which, unfortunately, only a pianist can operate. Some prefer keyboard controls because they offer a faster response time than paddle inputs, or they are easier to program, or this approach doesn't limit the market to an audience with expensive joysticks. I don't think the latter should influence your choice, but thought should be given to which method would make the game more enjoyable. Games that require considerable time to master the controls, often prove too frustrating to play.

Apparently, Apple owners like games which pit them against a competitive computer opponent. There are several multi-player games in which groups of two or more will simultaneously compete against each other. Most of these contests are sports or card games involving two or more players. The cooperative game is rarely seen, except in games where the computer com-

petitor is much too skillful. The arcade game "Ripoff" involves a computer opponent that is more than a match for two players playing simultaneously. It is the lone exception to the one-player-against-the-machine game.

So far, we have discussed theory and generalizations that should increase a game's playability and appeal to the public. Concrete examples of the more popular games should give you a much more solid foundation for your own designs.

EXAMPLE ARCADE GAMES

Space Invaders was the first really popular arcade game. It is a game wherein the object is to defend your turf against an alien horde of ferocious invaders that attack your castles and gun bases with a barrage of undulating bullets. It is actually a timed game, since you only have a limited amount of time to destroy the entire attacking wave before they descend to the ground in marching formation and overrun your lone gun base.

The elimination of each alien acts as a visual scorekeeping device. Although you can never win, only survive as long as possible (thus getting the maximum play time for your quarter), elimination of each attacking wave is an intermediate goal and a staving off of your inevitable doom. Each successive level becomes more difficult since the aliens, which begin their attack closer to Earth, limit the amount of time you have to destroy them. Their approaching proximity to your mobile gun base decreases your reaction time needed to avoid enemy fire.

Shoot-'em-up games like Sneakers, Galaxian, Threshold and Gamma Goblins are actually spin-offs of the Space Invaders theme. Whether they are set in space or on the ground, each has varieties of targets that are bent on your destruction. The targets or attackers are no longer static. Either they appear to dodge your fire, or they resort to kamikaze-type attacks.

The strong appeal of these types of games is based on curiosity and game depth. You are inspired to do better with each game just to see what the attackers are going to look like in the next level and what their tactics will consist of. The concept is variety, with each successive level slightly harder than the last. Although most offer an unlimited number of bullets, Threshold controls rapid, random, and wasteful firing by overheating your lasers. Thus, your firing must be more accurate and paced during the game.

The popularity of Pacman can be attributed to the game's design. First, it satisfies the fantasy concept of a person's childhood dreams. As children, they dreamt that they were being chased by evil monsters or ghosts, and felt powerless to stop them. They wished that there was some way to turn the tables, if only for a few moments. Pacman's four energy dots fulfill that fantasy. The game also offers the visual feedback of the number of remaining dots to be eaten at each level. And since clearing each individual level is an immediate goal, even beginners believe a level can be cleared. Because Pacman is

a game of consumption rather than one of destruction, it appeals to players of both sexes.

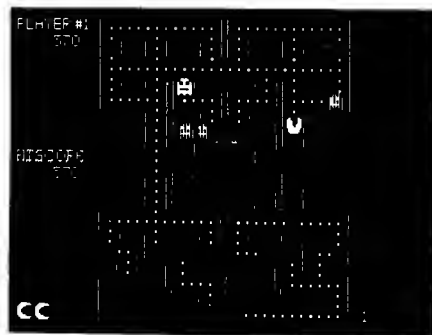
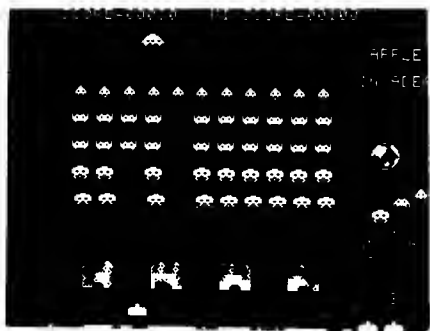
The game becomes a learning experience to the more advanced player, since the ghosts follow a discernible pattern rather than move randomly. A player is able to eventually predict their movements and consequently develop a technique to clear all the dots on a particular level. The long term goal is survival and the highest score. The game is designed so that you gain more pleasure as you get better. Thus, players are willing to devote the time and money to master the game.

Scrolling games, such as Scramble and Vanguard as played in the arcades, and Pegasus II on the Apple, wherein your ship travels over a multi-screen world, benefit strongly from player curiosity and visual variety. Vanguard, a shoot-'em-up game in which your ship is attacked by a variety of enemy vessels and creatures, has an extremely long sinuous tunnel with various types of chambers. The game has so many sections, combined with scrolling directions which change from horizontal to diagonal to vertical, that it is like playing many different arcade games at once. The player is given the option several times during the game to enter battle with a time-limited energized spacecraft which is equipped for ramming the enemy, or merely four plain old directional lasers. A map displayed at the lower corner informs the player of his progress. The curiosity factor is so enticing in this game, thirty seconds are provided to lure you into inserting another quarter in order to allow you to continue from where you left off with this unique form of arcade addiction.

The popularity of Pacman can be attributed to the game's design. First, it satisfies the fantasy concept of a person's childhood dreams. As children, they dreamt that they were being chased by evil monsters or ghosts, and felt powerless to stop them. They wished that there was some way to turn the tables, if only for a few moments. Pacman's four energy dots fulfill that fantasy. The game also offers the visual feedback of the number of remaining dots to be eaten at each level. And since clearing each individual level is an immediate goal, even beginners believe a level can be cleared.

The game becomes a learning experience to the more advanced player, since the ghosts follow a discernible pattern rather than move randomly. A player is able to eventually predict their movements and consequently develop a technique to clear all the dots on a particular level. The long term goal is survival and the highest score. The game is designed so that you gain more pleasure as you get better. Thus, players are willing to devote the time and money to master the game.

Scrolling games, such as Scramble and Vanguard as played in the arcades, and Pegasus II on the Apple, wherein your ship travels over a multi-screen world, benefit strongly from player curiosity and visual variety. Vanguard, a shoot-'em-up game in which your ship is attacked by a variety of enemy vessels and creatures, has an extremely long sinuous tunnel with various types of chambers. The game has so many sections, combined with scrolling directions which change from horizontal to diagonal to vertical, that it is like playing



many different arcade games at once. The player is given the option several times during the game to enter battle with a time-limited energized spacecraft which is equipped for ramming the enemy, or merely four plain old directional lasers. A map displayed at the lower corner informs the player of his progress. The curiosity factor is so enticing in this game, thirty seconds are provided to lure you into inserting another quarter in order to allow you to continue from where you left off with this unique form of arcade addiction.

Pegasus II, as implemented on the Apple, offers variety in terrain, targets and types of enemy. Besides trying to survive ground-launched rockets, a meteor field, attacking birds, and flying saucers, you must defeat a horde of laser-armed dragons that separate you from your refueling base. Your immediate goal is to reach the base before running out of fuel. This means accurate shooting, for enemies like dragons can delay your rendezvous with the base. Long term goals consist of reaching the tunnel and scoring the highest number of points.

In closing, I hope I have provided you with some acquired skills for creating your own visual masterpieces. The arcade versions described above are, as of this writing, being surpassed in quality by the dazzling array of games currently arriving on the personal computer market from talented graphics programmers.

My hope is that this book has provided some techniques and insights into graphics game design and programming; possibly even enough to allow you to join the ranks of successful Apple game designers.

INDEX

Addition & Subtraction, 45-46
Addressing modes, 42, 74, 112-114
AND instruction, 131-132, 209-210
Animation Apple Shapes, 26-29
Animation HPLOT Shapes, 78-81
Apple Shape Tables, 16-25, 81-85
Applesoft Hi-Res, 9, 29
Applesoft ROM, 69-71
ASL & LSR instructions, 53
Assemblers, 25
Assembly language, 36-46
Background fill, 14
Background preservation while drawing, 140-146
Bit-mapped Shape Tables, 100-109
Bomb drop, 154-157, 161-164
Branch instructions, 44-45
Breakout game, 51-68
Bullet motion, 157-160
Character generators, 30-33
Collisions, 209-212
Color problems, 123-127
Compare instructions, 43
Debug package, 204-205
Drawing bit-mapped shapes, 111-118
EOR instruction, 119-120
Explosions, 214-220
Game design & theory, 281-285
Graphic screen layouts, 9, 87
Graphic screen switches, 10
Hexadecimal numbers, 36-37
HI-RES color, 14, 89-92
HI-RES screen layout, 87-99
HPLOT shapes, 73-77

- Increment & decrement instructions, 43
- Interfacing bit-mapping to Applesoft, 135-139
- Invaders game, 164-181
- Joystick control, 152-153
- Laser fire, 205, 208
- Line memory address, 93-97
- Load instructions, 42
- Lookup tables, 111-112
- LO-RES graphics, 47-50
- Memory constraints, 11
- Memory map, 38-39
- Mountain background generator, 239
- Mountain collision test, 246-248
- Movement constraints & advantages, 132-133
- Odd / even test, 54
- OR instruction, 120
- Order of game events, 246
- Pac-Man, 283-284
- Paddle button trigger, 205
- Paddle crosstalk, 152-153
- Paddle routine, 147-151
- Page flipping, 15-16, 225-236
- Pegasus II, 285
- Print routine, 56-57
- Program Status Word, 39
- Raster shape tables, 100-109
- Scorekeeping, 55-56, 220-224
- Screen erase, 128-131
- Scrolling — vertical, 271-277
- Scrolling — horizontal, 278-280
- Scrolling games, 237-270
- Scrolling subroutine, 240-241
- Selective drawing control, 131-134
- Space Invaders, 283
- Space ship — steerable, 183-194
- Space ship — steerable & floating, 195-203
- Store instructions, 42-43
- XDRAWing bit-mapped shapes, 119-123



Jeffrey Stanton received a BME (1967) and a MSME (1969) from Rensselaer Polytechnic Institute. He worked as a control systems engineer and mechanical engineer for the aerospace industry in the early 1970's. His strong interest in computer game design sidetracked his career as a photographer and book illustrator in the late 1970's. Although he occasionally does a commercial assignment and owns a postcard company, much of his time is devoted to keeping abreast of the latest arcade game programming techniques on both the Apple and the Atari computers. He has several Apple games on the market and is writing a complex arcade game on the Atari 800. Jeffrey currently resides in Venice, California.

- Learn Apple Hi-Res Graphics from BASIC and machine language.
- Learn how to speed up your graphics.
- Learn raster graphics and bit mapping techniques.
- The only book to explain how to design arcade games from start to finish through the use of text, flow charts and working examples.
- Learn the theory of how to design a playable game.
- Requires a solid foundation in BASIC programming on the Apple II.

\$19.95